Чичев А.А., Чекал Е.Г.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 2. ФАЙЛОВЫЕ СИСТЕМЫ

Учебное пособие



МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Факультет математики, информационных и авиационных технологий
Кафедра «Телекоммуникационных технологий и сетей»

Чичев А.А., Чекал Е.Г.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 2. ФАЙЛОВЫЕ СИСТЕМЫ

Учебное пособие

Ульяновск
2021

УДК 004.052
ББК 32.973
Ч 78

Печатается по решению Ученого совета
факультета математики, информационных и авиационных технологий
Ульяновского государственного университета
(протокол № от)

Рецензенты:
заведующий кафедрой телекоммуникационных технологий и сетей УлГУ, д.т.н., профессор А. А. Смагин;
доцент кафедры «Прикладная математика и информатика» к.ф.-м.н., Т. Е. Родионова

Чичев А.А.
Ч78 Операционные системы. Часть 2. Файловые системы. Учебное пособие. / Чичев А.А., Чекал Е.Г. – Ульяновск: УлГУ, 2021. – с.

Учебное пособие составлено в соответствии с программой дисциплины «Операционные системы», и предусматривает подготовку инженеров и бакалавров по направлениям 09.03.02 «Информационные системы и технологии», 11.03.02 Инфокоммуникационные технологии и системы связи, 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем» и специальности 10.05.01 «Компьютерная безопасность». Может использоваться студентами родственных специальностей и направлений.

Пособие состоит из четырех частей. В части первой приведены краткие сведения о работе основных подсистем операционных систем, а также методические указания к лабораторным работам по установке, конфигурированию и эксплуатации операционных систем. Во второй (данной) части пособия рассматриваются устройство средств хранения, форматы разбиения, файловые системы: UFS/UFS2, клоны UFS (в т.ч. ext), fat12/16/32, ntfs, iso 9660. В третьей части пособия рассматриваются ЭМВОС, сетевые технологии и стандарты на них, стеки сетевых протоколов (SMB, IPX/SPX, TSP/IP, AppleTalk, SNA), именование сетевых объектов на различных уровнях ЭМВОС, взаимодействие процессов и сервисы. Часть четвертая содержит вопросы с ответами по данной дисциплине — ФОС.

Пособие предназначено для практического руководства при проведении преподавателями лекций и практических занятий и выполнении самостоятельных заданий студентами указанных направлений и специальностей всех форм обучения.

УДК 004.052
ББК 32.973

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Исторически сложилось так, что в настоящее время самой быстро развивающейся, самой функционально продвинутой, самой сложной алгоритмически, самой большой по объёму и вообще самой-самой операционной системой стала операционная система Linux. Возможно, кому-то это высказывание не нравится, но, увы, оно истинно.

Именно в развитие этой операционной системы вкладываются усилия тысяч высококвалифицированных разработчиков, как частных лиц, так и сотрудников фирм по всему миру. Объём проекта (Linux-5.x) оценивается в 20 (двадцать!) млн. строк кода — ещё никогда человечество не разрабатывало столь сложные и объёмные программные системы.

Именно эта операционная система обеспечивает функционирование самых мощных ЭВМ нашего мира — согласно списка Top-500 её используют более 95% суперЭВМ [16].

Именно над развитием этой операционной системы работают сотрудники крупнейших корпораций мира — IBM, HP, Oracle, Intel, Google, Samsung и других, и даже Microsoft (!). Ещё совсем недавно, в 90-ые годы, основной вклад в развитие Linux делали частные лица. Однако, в последние десятилетия в крупнейших ИТ-корпорациях мира появились подразделения, специализирующиеся на разработке в Linux и на её совершенствовании.

Именно эта операционная система совместно с другими Unix'ами является основой Интернета, а «Интернет — это наше всё».

Именно эта операционная система совместно с другими Unix'ами и Unix-подобными ОС является базой для построения высоконадёжных высокодоступных (24x7) информационных систем.

Именно эта операционная система наряду с другими Unix'ами используется на серверах корпораций, банков, бирж, является основой систем управления в энергетике (в том числе АЭС), на транспорте (диспетчерские системы), в связи (АТС, провайдеры Интернета и сотовой связи), в государственных структурах.

То есть, зависимость современной экономики от Unix'овых операционных систем (и прежде всего, Linux) не просто большая, а основополагающая: если вдруг завтра проснёмся, а Unix/Linux исчез (вот только вчера был — и нет его . . .), то мало не покажется — в 2008 году только один банк «лопнул» и это привело к мировому экономическому кризису, а если все и всё?

Специфической особенностью Linux является открытость исходных текстов ОС — лицензия GPL, по которой распространяется Linux, запрещает закрывать исходники. А это значит, что любой желающий разобраться в том, как устроена эта высокотехнологичная, высоконадёжная ОС, может это сделать. Тем более, что исходники Linux хорошо документированы. Конечно, разобраться в миллионах строк кода — это задача очень нетривиальная. «Порог вхождения» в

квалификацию очень высокий. Но возможность-то предоставляется!

По лицензии GPL распространяется не только сама ОС Linux, но и практически всё системное и прикладное ПО дистрибутивов Linux. А это десятки тысяч пакетов программ. Пример: репозитарий ALTLinux (Россия) содержит около 40 тысяч пакетов по состоянию на 2019 год.

А поскольку исходные тексты ОС открыты, то отсюда следует, что эта ОС сама по себе может являться учебным пособием по дисциплинам программирования, причём, учебным пособием очень продвинутым, написанным профессионалами. Аналогичными учебными пособиями являются и десятки тысяч пакетов, распространяемых также по лицензии GPL в составе дистрибутивов. И практически все программы и библиотеки имеют документацию — таково правило включения пакета в дистрибутив. Хотите стать профессиональным программистом? Нет проблем! Открывайте исходники linux-овых программ и самой ОС linux — и учитесь у профессионалов. То есть, действует основополагающий принцип педагогики: «Делай — как я!». Иначе говоря, для целей образования ОС Linux подходит очень хорошо: выполняется основной принцип образовательного процесса: «Делай как я! - делай лучше меня!». Реализуется основополагающее положение обучения по образцу.

Также следует сказать о том, что дистрибутивы Linux распространяются практически бесплатно — по лицензии GPL. То есть, реализуется иная модель бизнеса, нежели при распространении коммерческих дистрибутивов: заработок появляется не при продаже (перепродаже (спекуляции!) — дистрибьюторами) программы, а при сопровождении проданного, а это существенно более высокотехнологичное действие, требующее и гораздо больших знаний и квалификации, и большего количества людей. Сопровождение сложного ПО — это высокие и очень высокие технологии. Сопровождать сложное ПО — это намного сложнее, чем быть просто программистом. Научиться программировать не сложно, программистов миллионы, а разобраться в сложной большой программе — о-о! это нужна квалификация, это не каждый может. Недаром существует у программеров поговорка: «Чем в этой проге разобраться — легче новую написать!».

Следовательно, использование Linux способствует развитию высокотехнологичных отраслей экономики.

И, наконец, очень важный момент: если используется Российский дистрибутив Linux, то и деньги за его разработку, использование и сопровождение остаются в России, то есть, у нас с Вами. А это немалые деньги, ведь до сих пор (2019 год) Россия тратит на закупку импортного ПО до $5 (пяти) млрд в год.

Поэтому в данном пособии рассматривается в основном операционная система Unix (и прежде всего — Linux), а остальные упоминаются по мере необходимости в целях сравнения. И основным дистрибутивом, рассматриваемом в пособии, является ALTLinux. Почему именно ALTLinux — смотри ниже.

Пособие состоит из нескольких лекций. Количество лекций определяется объёмом дисциплины «Операционные системы», а это объём у разных специальностей разный: от 8 лекций (полсеместра) до 48 лекций (три семестра). В трёхсеместрой дисциплине «ОС» по теме «Файловые системы» читалось от 5 до 7 лекций в разные годы. Исходя из этого в данном пособии материал разбит на 7 лекций. В случаях меньшего объёма дисциплины «ОС» часть материала можно пропустить/сократить или вынести в практические занятия (семинары).

Обращаем внимание, что наиболее важный материал находится во 2-ой (форматы разбиения) и 3-ей (форматирование) лекциях.

Пособие предназначено для практического руководства при проведении преподавателями лекций, семинаров и лабораторных занятий и выполнении заданий студентами указанных специальностей всех форм обучения.

Учебное пособие составлено в соответствии с программой дисциплины «Операционные системы», и предусматривает подготовку инженеров и бакалавров по направлениям 11.03.02 «Инфокоммуникационные технологии и системы связи», 09.03.02 «Информационные системы и технологии», 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем» и специальности 10.05.01 «Компьютерная безопасность». Может использоваться студентами родственных специальностей и направлений.

(Настоятельно!) рекомендуемый авторами дистрибутив Linux для целей образования (и не только) — это ALTLinux (берётся отсюда: http://ftp.altlinux.ru). Обоснование:

а) это самый хорошо локализованный дистрибутив: не только хорошо локализованы много программ (хорошо локализованый — это переведены и меню программного интерфейса, и программная документация), но и имеется очень много русской документации практически по всем вопросам (например, heap.altlinux.ru или wiki.altlinux.ru); этот пункт, пожалуй, важнейший в обосновании выбора, ибо несмотря на напряжённый труд учителей английского языка в школах и преподавателей кафедр иностранных языков в ВУЗах, знание студентами английского крайне посредственное;

б) самая лучшая поддержка в России, в том числе, бесплатная сообществом ALTLinux на форуме (http://forum.altlinux.org), в рассылках (https://www.altlinux.org/MailingLists), IRC (https://www.altlinux.org/IRC), в социальных сетях (https://telegram.me/alt_linux, https://vk.com/altlinux, https://vk.com/simplylinux, https://www.facebook.com/groups/136328550579/, https://plus.google.com/communities/108911472444655347698) и прочих сервисах (https://www.altlinux.org/Contacts); среднее время ожидания ответа на вопрос — несколько часов; ни один другой дистрибутив linux, претендующий на «русскость», подобной скоростью похвастаться не может;

в) достаточно хорошая распространённость в России, в том числе, в школах;

г) это реально Российский дистрибутив — разработчики живут в РФ и репозитарий находится в России; причём **репозитарий ALTLinux — это настоящий отдельный**

самостоятельный самодостаточный репозитарий с соответствующей инфраструктурой, а не зеркало какого-то там;

д) включен в Реестр (https://reestr.minsvyaz.ru/) - на начало 2021 года это дистрибутивы: «Альт Образование», «Базальт Рабочая станция», «Альт Сервер», «Альт Линукс СПТ», «ОПЕРАЦИОННАЯ СИСТЕМА АЛЬТ 8 СП»: (https://reestr.digital.gov.ru/reestr/?sort_by=date&sort=asc&sort_by=date&sort=asc&class=&name=alt&owner_status=&owner_name=&class%5B%5D=54112&name=%D0%B0%D0%BB%D1%8C%D1%82&owner_status=&owner_name=&set_filter=Y);

е) дистрибутив ALTLinux существует/распространяется как в виде полных сборок (почти всё, что нужно в одном .iso), так и в виде заготовок для создания своих сборок — стартеркиты (StarterKit`s), что очень удобно для построения своих оригинальных систем; среди стартеркитов есть сборки и для слабых машин (и даже очень слабых) - https://www.altlinux.org/Starterkits/Memory

ж) ALTLinux, а ныне «Базальт СПО», проводят обучение работе с ОС «Альт» ИТ-специалистов, пользователей, преподавателей и просто жаждущих на различных курсах, семинарах (https://www.basealt.ru/courses/training/), конференциях, вебинарах (https://www.basealt.ru/courses/vebinary/) разного уровня сложности и портале дистанционной поддержки (https://kurs.basealt.ru/). Возможно создание сертифицированных учебных центров на базе вузов. На портале дистанционной поддержки содержится много обучающих курсов, в том числе, видеокурсов по ОС «Альт» и прикладным программам дистрибутива, а также методические материалы по их применению в учебном процессе. Обучение, в том числе и бесплатное, с выдачей сертификата;

з) «Базальт СПО» проводит ежегодную конференцию разработчиков свободных программ на базе Калужского ИТ-кластера и ежегодную конференцию «СПО в высшей школе» при поддержке Института программных систем РАН в городе Переславль-Залесский. Сборники тезисов конференций размещаются в **РИНЦ** и на **https://books.altlinux.org** Кроме того, «Базальт СПО» является соорганизатором ежегодной научно-практической конференции «OS DAY» (https://osday.ru/);

и) ALTLinux работает на российских процессорах Эльбрус — версии «Альт Рабочая станция» и «Альт Сервер» для отечественной вычислительной платформы «Эльбрус» (**http://mcst.ru/os-alt-na-platforme-elbrus-rossijskaya-os-na-rossijskom-processore**).

ЛЕКЦИЯ 1. БЛОКОВЫЕ УСТРОЙСТВА

1.1. Общие сведения об устройствах

1.1.1. Виды устройств

В данном пособии мы рассматриваем устройства ввода/вывода ЭВМ (вычислительных систем). Подобные устройства являются обязательным элементов вычислительных систем.

Устройства ввода/вывода бывают:

- только устройства ввода,
- только устройства вывода,
- и устройства с которых можно вводить информацию в ЭВМ и на которые можно выводить информацию из ЭВМ.

Пример устройства только ввода: клавиатура (в смысле, внешняя клавиатура). НО! Только очень старые клавиатуры были устройствами только ввода. Когда на клавиатурах появились лампочки, то они приобрели новое качество — появилась возможность выводить на клавиатуры некоторую информацию, как правило, сигналы о состоянии ЭВМ.

Пример устройства только вывода: динамики. Как правило, динамики только выводят информацию. НО! Не только лишь все динамики могут выводить информацию, иногда они могут и вводить информацию в ЭВМ, ЕСЛИ на ЭВМ будет установлена некоторая программа, которая сможет использовать динамики в качестве микрофона.

Иначе говоря, не всё так однозначно.

1.1.2. Символьные и блоковые устройства

Устройства ввода/вывода бывают:

- символьные,
- блоковые.

Критерием отнесения устройства ввода/вывода к символьным или блоковым устройствам является способ обмена информацией с этим устройством.

Если обмен информацией осуществляется побайтно (потоком байт, «посимвольно», хотя байт — это не только символ), то устройство относится к группе символьных устройств. Как правило, это устройства «без памяти»: вы прочитали байт, всё, в устройстве его больше нет, на его месте стоит уже следующий байт. При новой попытке чтения с устройства этот следующий байт и будет выдан, либо получим ошибку, если следующий байт ещё не сгенерирован. Однако, могут быть исключения. Например, видеокарта. Это символьное устройство и обмен с ней осуществляется потоком байт. Но это устройство имеет специфику: при обмене с видеокартой

используется видеобуфер, то есть, область памяти размером в разрешение экрана, в которой сохраняется отображаемая информация. И эта специфика — наличие видеобуфера, позволяет обращаться к нему в режиме произвольного доступа, то есть, можно получить доступ к конкретным пикселям или к некоторой области экрана («окну» программы). До тех пор, пока пиксели не затёрты обновлением отображаемой информации.

Если же обмен информацией с устройством осуществляется блоками, то есть, за один раз передаётся/принимается блок информации (128, 256, 512, 1024, 2048, 4096, . . ., 32768 байт), то устройство относится к группе блоковых устройств.

В данном пособии рассматриваются именно блоковые устройства ввода/вывода. Они интересны тем, что являются, как правило, устройствами долговременного хранения информации. Одновременно они являются устройствами, с помощью которых возможен перенос информации с одной ЭВМ на другую. При царе Горохе, когда сетей ещё не было, эта способность (функциональность) блоковых устройств широко использовалась.

Очевидно, что при реализации функции хранения информации на устройствах возникает ряд вопросов:

а) скорость записи/чтения информации,

б) надёжность хранения информации,

в) долговременность хранения информации,

г) совместимость устройства с ЭВМ, то есть, может ли вычислительная система опознать и использовать устройство,

д) разграничение доступа к информации, хранящейся на устройстве,

е) порядок доступа к информации, имеющейся на устройстве: последовательный или произвольный,

- и другие вопросы.

Вопросы а), б) и в) имеют отношение в основном к аппаратной (технической) реализации устройства.

Вопрос г) имеет отношение не только к аппаратной реализации устройства, но и к программной реализации вычислительной системы (ЭВМ), и прежде всего, есть ли необходимый драйвер для работы с этим устройством ввода/вывода в составе ОС вычислительной системы. Или хотя бы некоторая программа, способная работать с этим устройством ввода/вывода.

А, вот, вопросы д) и е) очень интересны. И решаются совсем не просто. Ибо требуют изменения самого подхода к технологии записи/чтения информации.

В основном, именно решение данных вопросов а)-е) и привело в итоге к тому, что появилось разделение всех устройств ввода/вывода на устройства символьные и блоковые. Вообще-то, идея использовать блоки информации при работе с устройствами появилась несколько ранее, нежели встали вопросы д) и е), и связано это было с необходимостью ускорить запись/чтения информации. Однако, во всей полноте и безальтернативности идея блоковой

записи/чтения была использована именно при решении вопросов д) и е).

И вопросы эти были решены посредством создания на блоковых устройствах (здесь звучат фанфары!) файловых систем.

Дело в том, что вопрос д) - «разграничение доступа к информации» - имеет два аспекта:

- как определить, какая программа писала информацию на устройство и, соответственно, какая программа будет читать и обрабатывать эту информацию; ведь, хотя сейчас множество программ ещё конечно (здесь «множество» понимается в математическом смысле), но тенденция такова, что это множество определённо превращается в счётное множество (какое множество называется счётным?); и ещё заметьте, что нынешние универсальные операционные системы, как правило, многозадачные;

- как определить, какой пользователь записал информацию и кто имеет право её читать; ведь нынешние универсальные операционные системы, очень часто, многопользовательские.

А решение вопроса е) в случае произвольного доступа требует определения специфической адресации имеющейся на устройстве информации. Причём сложность с адресацией заключается в том, что мы, люди, ориентированы на символьную адресацию (типа «Васёк», «Колян», «Вован», «Ульяновск», «Набережная Свияги», «УлГУ» - это последнее, правда, аббревиатура, но мы, люди, подобные аббревиатуры подсознательно переводим (развёртываем) в полное наименование, в полное имя). И для нас, людей, не представляет сложности по символьному имени представить (опознать) объект — легко, практически не задумываясь, переводим имя/адрес «Вася» в конкретного Васю. Но ЭВМ-то ориентирована на цифру, недаром она зовётся «вычислительной мафыной». И, соответственно, имена/адреса объектов в ЭВМ — цифровые. То есть, мы говорим «файл file.txt» (читается «файл_филЕ_точка_т_икс_т»), а ЭВМ, чтобы этот объект найти, должна перевести наше человеческое символьное имя в свое внутреннее цифровое имя, в адреса блоков, в которых этот «филЕ_точка_т_икс_т» находится. И обратите внимание, файлов на устройстве может быть много. И они могут кому-то (пользователю) принадлежать, и, возможно, не тому, кто хочет их почитать.

И в результате очевидно, что в случае произвольного доступа устройство ввода/вывода никак не может быть символьным — только блоковым. Это, ведь, не просто очевидно, это совершенно очевидно.

Вот в такой полной постановке вопросы д) и е) и были решены с помощью превращения устройств ввода/вывода в блоковые и создания на них файловых систем.

1.1.3. Блоковые устройства

Итак, устройство блоковое, если ввод/вывод на него реализуется блоками информации. Блоки информации пишутся в определённые места устройства и при этом запоминается, куда этот

блок информации был записан. Чтобы потом, при чтении, можно было эту информацию найти.

В идеале, зная адрес блока, можно сразу получить содержание этого блока — информацию из этого блока, не обращая внимание на остальные блоки. Или, при записи, записать информацию в указанный блок на устройстве, не обращая внимание на остальные блоки. Это означает, что блоковое устройство ввода/вывода поддерживает произвольный доступ к информации. Пример: винчестер, флешка, DVDROM. Отсюда следует интересный вывод: к блоковому устройству ввода/вывода с произвольным доступом могут иметь доступ одновременно несколько программ. Поскольку, программа, получив (или записав) блок информации, устройство освобождает — она же получила, что хотела. Далее запрос на чтение или запись блока информации может выдать другая программа. Иначе говоря, блоковое устройство ввода/вывода с произвольным доступом - «расшариваемое» устройство.

Но не все блоковые устройства работают таким образом. Существую блоковые устройства с последовательным доступом к информации. То есть, чтобы найти нужный блок, нужно предварительно прочитать предыдущие блоки, чтобы добраться до нужного. Пример: магнитные ленты. Как правило, алгоритмы работы таких блоковых устройств ввода/вывода с последовательным доступом проще, нежели блоковых устройств ввода/вывода с произвольным доступом. Кроме того, такие устройства, в силу используемого метода доступа — блоковый последовательный, как правило, используются в монопольном режиме. То есть, если некоторая программа ленту читает, то другая программа не сможет поиметь ленту до тех пор, пока первая программа не закончит работу с лентой.

1.2. Особенности винчестеров

Первое, что нужно помнить про винчестеры: **винчестер — самая важная часть компьютера.**

Все остальные запчасти при выходе их из строя легко заменить. Вы же ИТ-шники, вы же знаете, как разобрать и собрать комп так, чтобы ни одного лишнего болтика не сталось и он снова заработал, может даже ещё лучше. Ну, по крайней мере, должны это уметь. Если что-то в компе сломалось/сгорело: деньги — магазин (или youla.ru, avito.ru) — купил — принёс — заменил — всё снова работает. Ну, если системную плату заменили , то Винду, вероятно, придётся переустановить, а linux не обязательно.

Но если у вас «сдох» винт!? О-о! Это будет гора слёз, море костей и мешок крика, что все виноваты. То есть, наоборот. Да, неважно. Всё равно так. Ведь на винте остались ваши драгоценные и неповторимые шедевры — потеря невосполнимая.

Поэтому винт — самая драгоценная часть компа.

И не верьте всяким слухам о взрывающихся винчестерах, о вакууме внутри винчестера, о

вирусах, способных прожигать дырки в блинах и заставляющих резонировать головки, об увеличении оборотистости диска программным путем в 1.3 раза и прочим страшилкам «на ночь».

1.2.1. История вопроса

Жёсткий диск (HDD, винчестер) был впервые представлен достопочтенной публике в сентябре далёкого 1956 года компанией IBM. Он входил в комплект компьютера IBM 305 RAMAC (**RAMAC** - "*Random Access Method of Accounting and Control*"). На рисунке 1 на заднем фоне за оператором как раз и видим этот винчестер. Да-да, этот «шкаф» и есть самая первая версия винчестера. Он назывался «**350 Disk File».** По центру видим пакет блинов (50 штук, 24 дюйма диаметром) на шпинделе, справа от пакета блинов — устройство управления головками. Остальное содержание «шкафа» - сопутствующая механика и электроника. Вес устройства — около тонны.



Рис. 1. Первый винчестер.

Как и на современных винчестерах, блины (сплав на основе алюминия, иногда стекло, а в современных может быть и пластик) были двусторонними, то есть, были покрыты с двух сторон оксидом железа (то, что мы называем «ржавчиной»). Подобные блины (рыжие) от винчестеров ЕС ЭВМ до недавнего времени можно было видеть на балконах, они использовались в качестве широкополосных антенн ТВ.

Однако, головок на первом винте было только две, то есть, зараз мог обрабатываться только один блин. В отличие от современных винтов, в которых, как правило, головок столько, сколько поверхностей блинов. Причём, как и на современных винтах для перемещения и позиционирования головок использовалась система сервоконтроля — о ней ниже поговорим.

Скорость вращения блинов была 1200 оборотов в минуту. Плотность записи 2000 бит на квадратный дюйм. Разбиения блина на зоны ещё не было. Общая емкость накопителя составляла 5 000 000 символов, это примерно **4.4 Мбайт**. Да, именно 4.4 мегабайт.

Разработка винчестера 350 Disk File велась в течение нескольких лет в лаборатории в Сан-Хосе (Калифорния). Руководителями разработки были сначала Рейнольд Б Джонсон (**Reynold B. Johnson**), а затем Луис Стивенс (**Louis Stevens**). Как всегда, проект вышел из сметы и едва-едва, только благодаря настырности команды разработчиков, был закончен.

Лет 10 фирма IBM пыталась продавать этот винчестер. К середине 60-х было продано около десяти штук. Почему никто не покупал? А потому что, в те времена память в ЭВМ оценивалась сотнями байт, килобайтами. А 4.4 Мегабайта — это было столько, что никто не знал куда столько девать, не было задач и, соответственно, программ, для решения которых требовалось бы столько памяти. А многозадачных ОС в те годы ещё не было.

Кстати, жёсткий диск винчестером обозвали много позже, уже в начале 70-х, когда появился (опять фирма IBM была первопроходцем) жёсткий диск, похожий на современные.

1.2.2. Устройство винчестера (современного)

Винчестер или жесткий диск (HDD — Hard Disk Drive) устроен следующим образом (см. рис. 2): на шпинделе, соединенным с электромотором, расположен блок из нескольких дисков (блинов), над поверхностью которых находятся головки для чтения/записи информации. Форма головкам придается в виде крыла и крепятся они на серпообразный поводок. При работе головки «летят» над поверхностью блинов в воздушном потоке, который создается при вращении этих блинов. Очевидно, что подъемная сила зависит от давления воздуха на головки. Оно же, в свою очередь, зависит от внешнего атмосферного давления. Поэтому некоторые производители указывают в спецификации на свои устройства предельный потолок эксплуатации (например, 3000

м). Диск разбит на дорожки (или треки), которые в свою очередь поделены на сектора. Две дорожки, равноудаленные от центра, но расположенные по разные стороны блина, образуют цилиндр.



Рис. 2. Устройство винчестера.

Корпус винта закрывается почти герметично, образуется гермозона — внутренность винта. Гермозона заполнена обычным обеспыленным воздухом под обычным атмосферным давлением. Для выравнивания давления внутри/снаружи в корпусе есть отверстие, диаметром 1-2 мм. Около него наклейка «No covered» - «Не закрывать». Под отверстием — фильтр тонкой очистки воздуха.

То есть, жесткие диски не герметичны. Кроме накопителей со сверхбольшими емкостями. В них гелий вместо воздуха, так как он гораздо менее плотный и создает меньше проблем в случае с большим количеством блинов внутри.

К нижней части винта крепится плата контроллера (см. рис. 3).

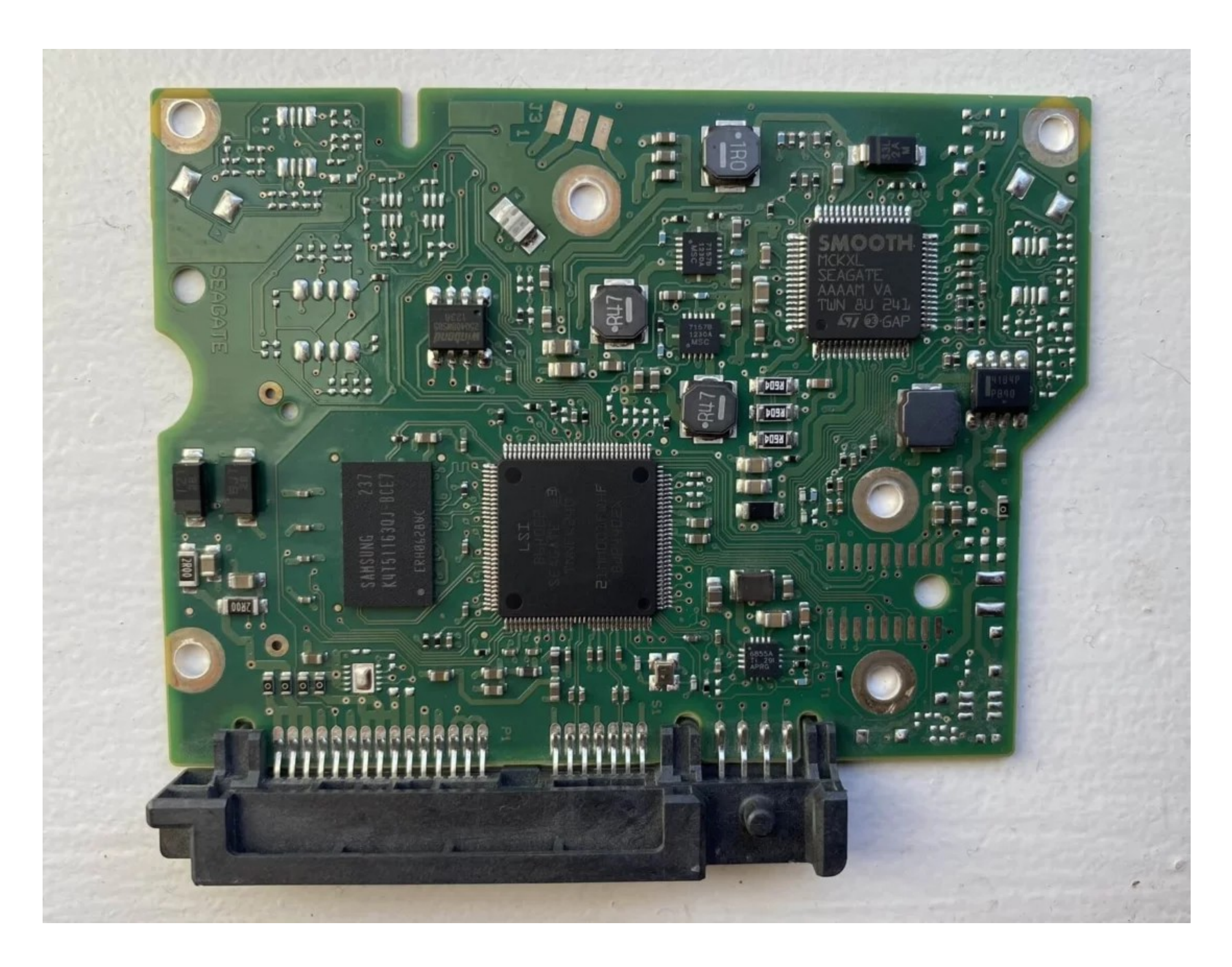


Рис.3. Плата контроллера — винт Seagate 3 Tb.

Что находится на плате контроллера. Стоит обратить внимание на 4 микросхемы:

- LSI B64002: по центру большая микросхема, это главный процессор (CPU, достаточно мощный), который обрабатывает инструкции, поток данных, занимается исправлением ошибок и т. д., то есть, выполняется основной алгоритм винта,
- Samsung K4T51163QJ: микросхема слева от CPU, это ОЗУ DDR2 SDRAM с 64 Мб памяти, с тактовой частотой 800 МГц, для кэширования данных,
- Smooth MCKXL: микросхема справа вверху, микроконтроллер, который управляет электромотором (шпиндельным двигателем),
- Winbond 25Q40BWS05: Небольшая микросхемка в центре, это 500 Кб флэш-памяти для хранения прошивки контроллера винта — программного обеспечения контроллера.

Компоненты печатной платы отличаются у разных марок и разных производителей, но несущественно. Винчестеры больших объемов требуют большего объема кэша (до 256 Мб DDR3 в последних «монстрах»), а CPU контроллера может быть несколько более сложным в плане обработки ошибок.

То есть, контроллер HDD — это достаточно мощная ЭВМ, работающая в режиме реального времени.



Рис. 4. Винт со снятой крышкой.

Внутреннее содержание винта показано на рис. 4. Желтый квадрат обозначает металлическую крышку, которая удерживает пластину с электродвигателем привода шпинделя, задача последнего — вращать блины. Конкретно в этом винте они вращаются со скоростью 7200 об/мин, но другие модели могут работать медленнее или быстрее. Более медленные накопители обладают не только низким уровнем шума и энергопотреблением, но и более низкой производительностью, в то время как более быстрые приводы могут достигать скорости 15 000 оборотов в минуту с соответствующим повышением скорости доступа. И шумом. Да ещё и греются сильней.

Дабы уменьшить вредное воздействие пыли и влаги из воздуха, рециркуляционный фильтр (зеленый квадрат - пылевлагопоглотитель) собирает мельчайшие частицы и задерживает их внутри себя. Воздух, перемещаемый вращением пластин, обеспечивает постоянный поток через фильтр.

Около верхней части блинов (в левом и правом углу от блинов , на 10 и 2 часа), а также рядом с рециркуляционным фильтром находятся три разделителя блинов: все они помогают уменьшить вибрацию, а также регулируют поток воздуха.

В левом верхнем углу изображения, обозначенном синей рамкой, находится один из двух постоянных стержневых магнитов, очень сильных. С точки зрения мусорщиков-утилизаторов

винтов — эти магниты являются самой ценной частью сдохшего винта. Они обеспечивают магнитное поле, необходимое для перемещения компонента, выделенного красным цветом — блока головок. Желтоватое устройство около конечной части головок — зона парковки. То есть, в этой модели зона парковки находится за пределами блинов.



Рис. 5. Винт со снятым магнитом и снятыми блинами.

Металлические пластины блока головок (см. рис. 5) — это рычаги перемещения головок, которые удерживают головки чтения/записи жесткого диска. Они двигаются по поверхностям блинов (сверху и снизу) с высокой скоростью.

Когда диск вращается, над поверхностями блинов поток воздуха создает подъемную силу, тем самым поднимая головку с поверхности. Образуется зазор всего лишь в 0,0000002 дюйма или менее 5 нм, на такой высоте головки «парят» на поверхностью блинов. Иначе говоря, помимо всего прочего, головки также рассчитываются с помощью правил аэродинамики (как самолёты).

Если головки поднимутся выше положенного, то они не смогут обнаружить изменения магнитных полей на поверхности блина. Или если они опустятся слишком низко (слишком близко к блину), то не просто покрытие блина может поцарапаться. Возникнет трение и головки очень быстро сгорят. Это основная причина, почему воздух внутри корпуса диска должен быть отфильтрован и быть нормального атмосферного давления: пыль и влага на поверхности диска просто сломают головки, а повышенное или пониженное атмосферное давление поднимут или опустят головки над поверхностью блина. Поэтому фирмы-производители в паспортных данных оговаривают, что винт такой-то модели работоспособен только при определённом атмосферном

давлении. Вывод: в горы с ноутбуком, в котором обычный винчестер — нельзя.

И на каждом рычаге, вообще-то, по две головки: записи и чтения. Они идут именно в такой последовательности почти рядом. Запись информации осуществляется с помощью головок, выполненных по технологии тонкопленочной индукционной системы (TFI), а чтение — с помощью головок, реализованных по технологии туннельного магниторезистивного устройства (TMR). Почему головок две? Потому что, требования к ним очень разные: головка записи должна создавать достаточно сильное магнитное поле, чтобы суметь перемагнитить поверхность блина, то есть, провод толстый, токи большие; а головка чтения должна создать достаточно большую ЭДС, чтобы сработали микросхемы и опознали информацию, то есть, провод тонюсенький и витков очень много (ЭДС, электродвижущая сила — что это такое, смотрите в курсе физики, раздел «электричество). Либо, если головка магниторезистивная, то очень маленький резистор (сопротивление), который изменяет сопротивление в зависимости от величины магнитного поля.

В последние десятилетия существенно изменились технологии, лежащие в основе магнитной дорожки и головок чтения/записи. Используются все более узкие и плотные дорожки, а головки становятся всё меньше в размерах, что в конечном итоге приводит к увеличению плотности записи на блинах и, соответственно, к увеличению объема винтов.

В целом, за последние десятилетия со скоростью доступа к информации на винчестерах практически ничего не поменялось, по-прежнему скорость случайного доступа составляет 5-10 мс. Основные причины слабого прогресса — ни скорость вращения блинов, ни скорость перемещения головок в требуемое положение практически не выросла за последние лет тридцать.

А, вот, технологии, лежащие в основе магнитной дорожки и головок чтения/записи в последние десятилетия существенно изменились. Используются все более узкие и плотные дорожки, а головки становятся всё меньше в размерах, что в конечном итоге приводит к увеличению плотности записи на блинах и, соответственно, к увеличению объема винтов. В настоящее время этот уровень примерно следующий. Толщина алюминиевого блина составляет около 1 мм. Его полировка такова, что средняя высота по всей поверхности составляет менее 30 нм. Далее на диск наносится никель-фосфорная грунтовка толщиной примерно 10 микрон. Затем вакуумным напылением наносится магнитный слой на основе сплавов кобальта из зёрен сложной формы (либо вытянутые овалы, либо кольца — у разных производителей технологии разные). Напыление производится в сильном магнитном слое, чтобы зерна магнитного слоя ориентировались определённым образом, например, вытянутые овалы — перпендикулярно поверхности блина. Сверху магнитный слой защищается тонким слоем углерода и очень «склизким» пластиком толщиной 10-12 нм. И снова полируется.

Bad-блоки. Современный уровень технологии не позволяет создать идеальный диск. Даже при самом тщательном наблюдении за процессом изготовления блинов, когда используются новейшие технологии и суперсовременное оборудование, на дисках появляются участки, где

процесс записи и чтения происходит с ошибками или же не происходит вовсе. Поэтому еще при изготовлении, вернее, сразу после него, производитель тщательно тестирует диск на специальном оборудовании в специальном технологическом режиме. Так как производство винчестеров является крупносерийным, естественно, данная процедура максимально автоматизирована. В результате тестирования становится доступной информация о дефектных участках, которая записывается в таблицу дефектов или дефект-лист (defect list, hardbad-list). Последний содержит адреса участков поверхности, непригодных для дальнейшего использования (тех самых bad-ов). Так как это очень важная информация, используемая на протяжении всего срока эксплуатации винчестера, то дефект-лист на диске представлен в нескольких копиях.

После того, как процесс тестирования успешно завершен, производится переадресация секторов, в результате чего сбойные секторы пропускаются и остаются вовсе незадействованными. Поэтому на новом диске создается видимость «безгрешной» поверхности.

Во второй дефект-лист (softbad-list) заносятся адреса запорченных участков, которые появились непосредственно во время эксплуатации жесткого диска. С помощью его можно судить, каково сегодняшнее состояние поверхности диска. Если он начал заполняться, то есть, контроллер обнаружил поврежденные участки или секторы и указал их адреса в дефект-листе, значит, процесс пошел. Правда, предугадать его интенсивность весьма не просто. Ознакомиться с содержанием softbad-list можно, посмотрев показания S.M.A.R.T. (Self-Monitoring Analysis and Reporting Technology).

Кстати, могут существовать еще два дефект-листа: лист сервометок и временный. Сервометки были разработаны для лучшего позиционирования головок, когда плотность записи достигает такой величины, что головки не могут быстро и точно перемещаться с одной дорожки на другую. Но сервометки тоже могут содержать ошибки. И для большей надежности винчестеров дефектные сервометки начали заносить в специальный, предназначенный только для них список.

И последний дефект-лист - временный, предназначенный для записи подозрительных, с точки зрения контроллера HDD, секторов. Например, если не удалось с первого раза считать или записать данные в определенный сектор, либо же время записи или чтения показалось контроллеру уж больно долгим (то есть, вышло за определенные рамки). Тогда контроллер заносит адреса подозрительных секторов во временный дефект-лист. Если с проблемными секторами он ничего не может сделать (ни записать, ни считать данные), то эти секторы фиксируются в softbad-list-e, и считаются bad-ами. Однако, контроллер просто так не заносит секторы в дефект-лист. Он придерживается пословицы - семь раз проверь, один раз запиши в дефект-лист. И не подумайте, что контроллер такой неторопливый. А лучше представим, какова будет емкость накопителя, если при малейшем подозрении на bad, контроллер будет отправлять адрес этого сектора в дефект-лист. Ведь когда сектор появился в любом из этих листов (кроме временного), он перестает существовать.

И каждый раз, когда на поверхности появляется bad-сектор, контроллер присваивает адресу

испорченного сектора адрес сектора с резервного места (из резервных дорожек — см. п. 1.2.5.). И при следующем обращении по этому адресу головки следуют к резерву и работают с переназначенным сектором. Единственный недостаток такого метода (называемого remap-ом) заключается в том, что в этом случае несколько уменьшается скорость работы винчестера.

1.2.3. Хранение информации

Винчестер — блоковое устройство, следовательно, информация на нём сохраняется фиксированными порциями, называемыми блоками (секторами). Блок для винчестера наименьшая обрабатываемая порция данных. И поскольку винчестер — блоковое устройство с произвольным доступом к данным, то блок должен иметь адрес, по которому его можно найти. Размер блока для старых винчестеров с форматом разбиения PC BIOS почти всегда равен 512 байт, а для новых винтов с форматом разбиения GPT — 4 кб. О форматах разбиения — ниже.

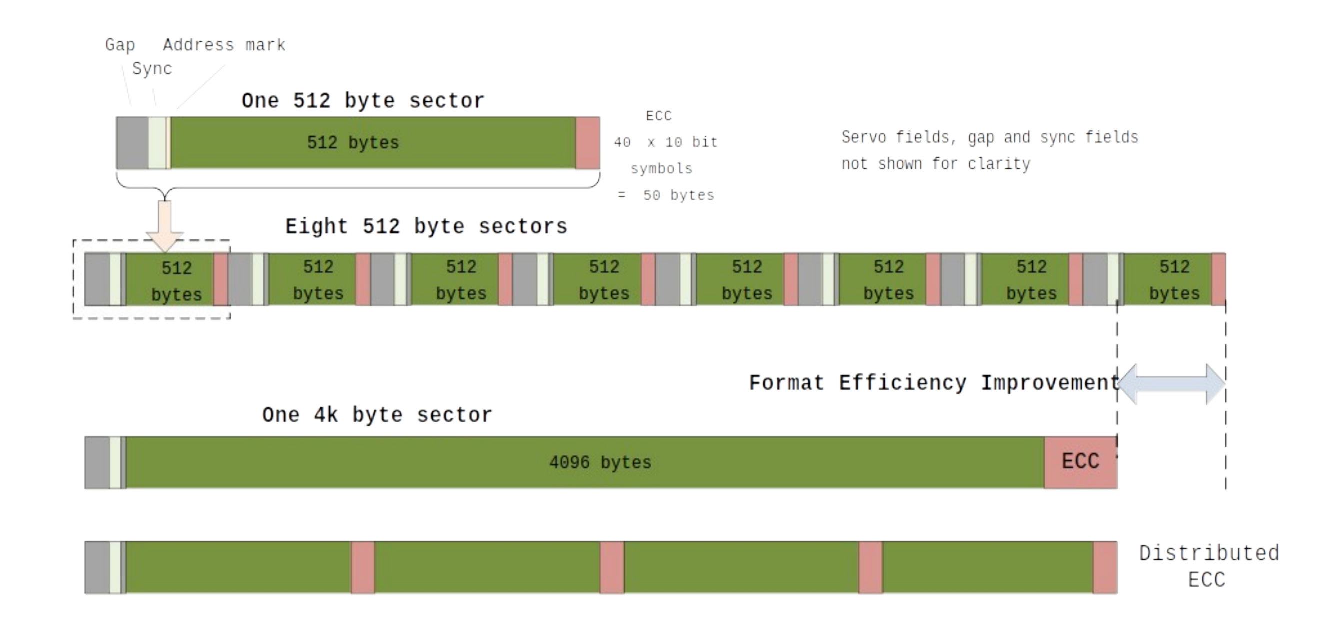


Рис. 6. Структуры секторов.

1. Блоки и сектора.

Понятие «блок» - логическое. А физически блок размещается в секторе на блине (см. рис. 6). Традиционный (512 байтный) сектор диска имеет следующую структуру:

1) Интервал: промежуток между секторами.
2) Код синхронизации: метка синхронизации, обозначающая начало сектора и позволяющая синхронизировать работу контроллера диска.

3) Метка адреса: метка, содержащая данные для идентификации номера и расположения сектора. В ней также хранится информация о состоянии сектора: 1) + 2) + 3) = ~15 байт.
4) Область данных: в этой области хранятся данные пользователя.
5) Область исправления ошибок: в этой области хранятся коды исправления ошибок, с помощью которых исправляются и восстанавливаются данные, которые могли быть повреждены во время чтения или записи ~ 50 байт.

То есть, эффективность использования сектора составляет 512/(512+15+50) = ~88%.

Этот низкоуровневый формат используется в винтостроении уже многие годы. Однако в связи с ростом емкости жестких дисков размер сектора неизбежно становится конструктивным ограничением для дальнейшего повышения емкости дисков и эффективности исправления ошибок. К примеру, если соотнести размер сектора с емкостью старых и современных дисков, то можно увидеть, что разрешение сектора многократно уменьшилось. Разрешение сектора (отношение размера сектора к общей емкости диска, выраженное в процентах) в значительной мере сократилось и, как следствие, стало неэффективным (см. таблицу 1).

Таблица 1. Разрешение сектора.

Емкость	Общее количество секторов	Разрешение сектора
40 МБ	80 000	0,001 %
400 ГБ	800 000 000	0,0000001 %
12 ТБ	24 000 000 000	0,000000004 %

Низкое разрешение подходит для управления небольшими разрозненными последовательностями данных. Однако современные приложения, как правило, оперируют блоками данных, которые намного больше размера сектора 512 байт.

И что еще более важно, небольшие сектора размером 512 байт занимают все меньшую площадь поверхности диска по мере повышения плотности записи. Это становится проблемой в контексте исправления ошибок и вследствие дефектов покрытия. На рис. 7, например, данные в секторе жесткого диска занимают меньшую площадь, что делает исправление ошибок сложнее, так как дефекты покрытия, имеющие прежний размер, повреждают больший процент данных и для их восстановления требуются более совершенные средства.

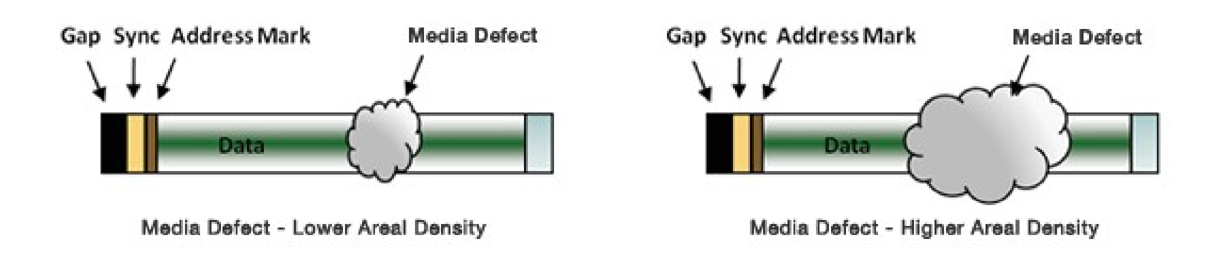


Рис. 7. Дефекты носителя и плотность записи

В секторе размером 512 байт, как правило, можно исправить дефект длиной до 50 байт. Современные жесткие диски с наибольшей плотностью записи практически достигли предела в

области исправления ошибок. Поэтому для дальнейшего развития средств исправления ошибок и повышения эффективности жестких дисков актуален стал переход к секторам большего размера.

И в декабре 2009 года был создан и утвержден новый формат сектора - «Advanced Format», размером 4 КБ.

В краткосрочной перспективе преимущества нового формата были не слишком заметны конечным пользователям, потому что новый формат не привел к моментальному увеличению емкости. Тем более что, старые программы по-прежнему использовали размер блока 512 байт. Поэтому в контроллерах новых дисков был дополнительно реализован алгоритм эмуляции секторов - преобразования данных из нового формата с размером сектора 4 КБ, используемого новыми дисками, в традиционный формат с размером сектора 512 байт, используемый старыми программами и ATA-контроллерами системной платы.

Однако в долгосрочной перспективе переход на секторы размером 4 КБ позволил увеличивать плотность записи данных и емкость жестких дисков, а также повышать надежность исправления ошибок.

В новом формате под интервал, код синхронизации и метку адреса отводится столько же места, сколько и раньше (~15 байт), а код исправления ошибок увеличен до 100 байт, что обеспечило повышение эффективности исправления ошибок и устойчивости к мелким частицам и дефектам поверхности. В результате эффективность нового формата увеличивается до 97% (4096 / (4096 + 115)), то есть почти на 10%.

Особенно важна поддержка нового формата сектора средствами операционных систем, поскольку именно модули ОС работают с дисками, транслируя запросы прикладного ПО. В частности, в linux поддержка дисков нового формата в ядре ОС реализована начиная с версии 2.6.31. Поддержка разбиения на разделы и форматирования дисков нового формата реализована в следующих дополнительных средствах Linux:

- Fdisk: GNU Fdisk — инструмент командной строки для разбиения жестких дисков на разделы. Начиная с версии 1.2.3 поддерживаются диски нового формата.
- Parted: GNU Parted — это графическое средство для разбиения жестких дисков на разделы. Начиная с версии 2.1 поддерживаются диски нового формата.

2. Зоны.

Та часть поверхности блина, на которую производится запись информации, называется зоной (см. рис. 8). На старых винтах на каждой поверхности блина была одна зона, однако с повышением требований к плотности записи, зон стали делать несколько, на современных винтах количество зон доходит до двадцати. В каждой зоне свое количество секторов на дорожке. Делается это для того, чтобы выровнять условия записи на дорожках, находящихся на разных расстояниях от центра блина и тем самым эффективнее использовать поверхность.

Наличие зон можно увидеть, задав тест линейного чтения винчестера (исправного). График будет выглядеть в виде нисходящих ступенек — скорость чтения в разных зонах — разная.

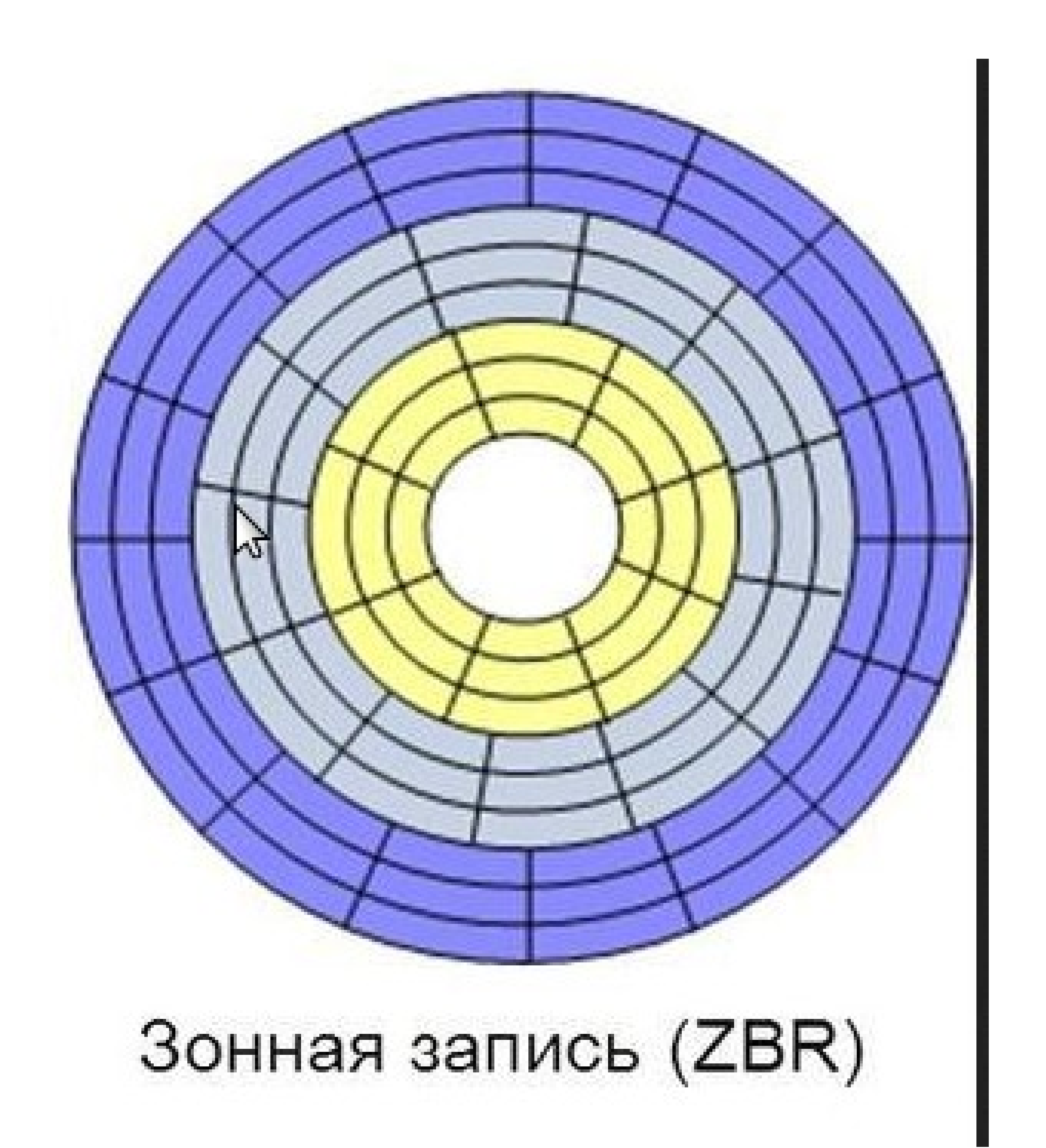


Рис. 8. Зоны на блинах.

Также в современных винчестерах, применяется эффективный, хотя и сложный в реализации метод PRML (Partial Response Maximum Likelihood) - метод максимального правдоподобия при частично перекрывающемся отклике от соседних дорожек). Дорожки расположены так близко, что головка считывает сигналы от нескольких соседних дорожек сразу, а затем на основе методов теории вероятностей выделяется сигнал от нужной дорожки по критерию максимума функции правдоподобия, которая на лету вычисляется CPU контроллера. При этом плотность записи повышается на 40-50%.

1.2.4. Адресация информации

Изначально адресация блоков (секторов) на жёстких дисках (а потом винчестерах) была геометрической. Ну, потому что очевидно же. Вы же учили в школе раздел математики — геометрия называется.

В далёкие 70-80-е года прошлого тысячелетия, для доступа к винчестеру надо было знать (и указывать) его "геометрию". Под "геометрией" понимается "физическое" количество цилиндров (дорожек) ("**C**" - от Cylinders), головок ("**H**" - от Heads), и секторов на дорожку ("**S**" - от Sectors). То есть, любой блок информации на винчестере характеризовался тремя переменными: C, H и S (отсюда - *CHS-адресация*). И надо отметить, что эти величины всегда были "реальными". При этом цилиндр и головка считаются от нуля, а сектор — от единицы. То есть, первый сектор диска в формате CHS будет иметь адрес (0, 0, 1). Почему адрес первого сектора (0,0,1), а не (0,0,0)? Потому, что сектор (0,0,0) занят меткой начала дорожки: она же круглая и на ней нужно где-то определить начало.

Однако, в те далёкие времена не подозревали, что совсем в скорости ёмкость винчестеров будет измеряться гигабайтами, десятками гигабайт, потом сотнями гигабайт, а ныне уже терабайтами и это ещё не предел. Ведь самый богатый в мире человек (и самый проклинаемый компьютерной братией в одном флаконе;) тоже как-то рассуждал "о бесконечности" 640кБ оперативной памяти. Соответственно, программисты BIOS программировали этот самый BIOS в соответствии со своими представлениями, которые оказались очень далёкими от реальности.

Что привело к возникновению «проблем больших дисков»:

134Mb, год 1990-й,

528Mb, год 1993-й,

2.11Gb, год 1996-й,

2.15Gb, год 1996-й,

3.28Gb, год 1996-1997-й,

4.23Gb, год 1997-й,

7.93Gb, год 1997-1998-й,

8.46Gb, год 1998-й,

33.8Gb, год 1999-й, лет,.

65.5Gb, год 2000-й, зима,

137.4Gb, год 2002-й.

Видите, какой пользователь всё-таки терпеливый? Но даже у этого терпеливого пользователя терпение кончилось и был сменён принцип адресации блоков на винтах с геометрического на логический: все сектора с самого первого сектора на самой внешней дорожке и до самого последнего сектора на самой внутренней дорожке были пересчитаны по порядку и порядковый номер сектора был назван его адресом. Эту адресацию секторов назвали LBA - Logical block addressing. И теперь объём (адресное пространство) винта можно представлять в виде отрезка прямой от адреса

- LBA 0 = Цилиндр 0/Головка 0/Сектор 1

- до адреса «который_указан_на_наклейке_винта» и определяет количество секторов на винте.

Современные винты работают с адресацией LBA. Но внутри контроллер винта пересчитывает адрес LBA по своему, строго индивидуальному для каждого экземпляра винта алгоритму снова в CHS, то есть, внутри винта работает снова геометрическая адресация. Но о ней знает только контроллер винта и никому о ней не сообщает.

Винчестер (или DVDROM, или иное устройство), способный поддерживать режим LBA, сообщает об этом в информации идентификации привода при опознании устройства.

Еще одно преимущество метода адресования LBA — то, что ограничение размера диска обусловлено лишь разрядностью LBA. В настоящее время для задания номера блока используется 48 бит, что при использовании двоичной системы счисления даёт возможность адресовать на приводе (2^{48}) 281 474 976 710 656 блоков (то есть, при блоке в 512 байт, 128 Петабайт). А при блоке 4 кб?

1.2.5. Служебная информация

Часть информационной поверхности блина используется накопителем для собственных нужд. Это область служебной, как ее еще иногда называют, инженерной информации. Она скрыта от пользователей и становится доступной при переводе винчестера в специальный технологический режим, осуществляемый при помощи стендового оборудования и особых утилит. Расположена эта информация на специально предназначенных для этого служебных цилиндрах, недоступных в пользовательском режиме работы HDD. Запись ее производится с помощью технологических команд в процессе изготовления винта.

Служебная информация делится на несколько типов:

- сервометки, предназначенные для стабилизации скорости вращения блинов, поиска секторов и точной установки головок на дорожки; в современных HDD на все блины пишутся специальные сервометки - поля, позволяющие сервоприводу по прочитанному с них сигналу получать информацию о текущем положении головки; по этим же сервометкам стабилизируется скорость вращения диска путем контроля периода следования сигнала; сервометки наносятся не только на места самих треков, но и в промежутках между ними; запись сервометок производится при изготовлении HDD на специальном устройстве, обеспечивающем внешнее позиционирование блока головок - серворайтере (Serwowriter); запись или восстановление сервометок средствами самого HDD (без внешнего позиционирования) невозможны;

- информация, служащая для адресации секторов с данными пользователя и контроля целостности этих данных — метки начала и конца секторов и коды контроля; включает в себя поля идентификатора сектора (дескриптора), служашего для идентификации сектора, и поля контрольной суммы (ECC check bytes), предназначенных для контроля целостности информации, записанной в поле данных сектора; это и есть то, что обычно понимают под низкоуровневым форматом;

- рабочие программы (микрокод), предназначенные для управления работой всех систем накопителя и обеспечивающих выполнение функций HDD; их копия хранится в микросхеме флэш-памяти на контроллере винта;

- паспорт винчестера, в котором записана информация о количестве дисков, головок, название фирмы-производителя и модели накопителя, дата его изготовления, страна изготовитель, номер конвейера, номер рабочей смены и многое другое; здесь же хранится и уникальный серийный номер винчестера;

- таблица дефектных секторов (аппаратный bad-list), служащая для аппаратной подмены сбойных участков поверхности из резерва. Эта информация используется электроникой и программным обеспечением винчестера в процессе работы и является важнейшей его частью, без которой физически полностью исправный накопитель был бы бесполезным куском железа;

- запасные дорожки, используемые для подмены сбойных секторов в соответствии с таблицей дефектных секторов (аппаратный bad-list);

- запасное место, используемое для подмены сбойных секторов, возникших в процессе эксплуатации (в соответствии с soft bad-list), для хранения самого soft bad-list, для работы системы S.M.A.R.T. (Self Monitoring Analysis and Reporting Technology).

1.2.6. Интерфейсы и порядок подключения

Во-первых, напомним определение интерфейса:

Интерфейс — набор правил и соглашений, определяющих взаимодействие между нижестоящим и вышестоящим уровнями (элементами, объектами) в системе (в том числе, вычислительной). **Интерфейс всегда предоставляет нижестоящий уровень вышестоящему.** Формально, интерфейс — это документ, в котором этот набор правил и соглашений описан. А практически он должен быть явно или неявно реализован (аппаратно, или программно, или аппаратно-программно).

Пример 1. Взаимодействие между сотрудником (подчинённым) и руководителем (начальником): сотрудник предоставляет руководителю интерфейс, по которому руководитель имеет сотрудника. Этот интерфейс определяется тремя параметрами: уровнем образования сотрудника (дипломом, например), квалификацией сотрудника (опытом работы) и должностной инструкцией. Руководитель может управлять сотрудником только в рамках этого интерфейса. Что произойдёт, если руководитель выдаст ценное указание сотруднику за пределами этого интерфейса?

Пример 2. Автомобиль предоставляет водителю интерфейс в виде совокупности устройств: руль, педали газа и тормоза, ручка КПП, панель приборов и прочая. С помощью этой совокупности элементов интерфейса автомобиля водитель им (автомобилем) управляет. Что произойдёт, если водитель попытается управлять автомобилем с помощью, например, «языка жестов», как на смартфоне?

Пример 3. Платы, вставленные в системную (материнскую) плату:

видеокарта — предоставляет интерфейс доступа к дисплею (AGP, PCI-ex или иной),

сетевая карта — предоставляет интерфейс доступа к сети (подуровень MAC канального уровня),

модем — предоставляет интерфейс доступа к телефонному каналу (аналоговому).

Пример 4. Интерфейс подключения HDD к вычислительной системе. Интерфейс предоставляет жёсткий диск, точнее, его контроллер. В этом интерфейсе определяются команды, которые могут восприниматься контроллером как правильные и отклики, порождаемые контроллером после отработки соответствующей команды. Со стороны вычислительной системы на системной плате ПЭВМ находится контроллер канала, который реализует интерфейс ATA. А со стороны устройства интерфейс реализует контроллер устройства. Что произойдёт, если вычислительная система выдаст контроллеру HDD что-то (типо, команда такая), что не описано в интерфейсе?

Далее рассматриваем только интерфейсы, используемые в ПЭВМ. В большинстве случаев в ПЭВМ используются винчестеры IDE (*Integrated Drive Electronics* — «электроника, встроенная в привод»), то есть, жёсткие диски, у которых снизу прикреплена плата контроллера. Это

интерфейсы ATA нескольких уровней. Некоторые из них уже устарели, сейчас используются параллельные UDMA-33, UDMA-66, UDMA-100, UDMA-133 и последовательный SATA. Числа в именах интерфейсов — максимальная скорость обмена в мегабайтах в секунду в канале (по кабелю), почти никогда недостижимая. Почему? Потому, что скорость определяется не только скоростью канала «контроллер IDE — контроллер диска».

1) **PATA**, в народе чаще называемый IDE. Он достаточно сложный, но у него есть плюсы и минусы. Во-первых, интерфейсов IDE много, как минимум, штук шесть: от очень древних (PIO и др.), до более современных UDMA-33 (40-pin-овый кабель) и UDMA-66/100/133 (80-pin-овый кабель). Во-вторых, на hdd , помимо интерфейсной колодки (40-штырьковой) и колодки питания (4-штырьковой), как правило, есть ещё одна колодка на 4/6/8/10 контактов а зависимости от марки hdd и производителя. На этих контактах нужно правильно поставить перемычку (определить статус устройства), иногда, не одну. Как ставить, обычно написано на наклейке на hdd. В третьих, на кабеле — три колодки: одна — в системную плату, а на две других можно подключать устройства hdd, DVDROM или что у вас там.

То есть, кабелей два (контроллер IDE (контроллер канала) на системной плате — двухканальный), а устройств можно подключить всего четыре. Причём так, чтобы статус у всех устройств был разный, иначе комп загружаться не будет. Статусы: IDE-0 (первый кабель) — master-1 и slave-1; IDE-1 (второй кабель) — master-2 и slave-2.

Определение статусов. Если интерфейс UDMA-33 (40-pin`овый кабель), то перемычки нужно поставить так, чтобы на каждом кабеле одно устройство было master, а второй — slave. Повторяю: на кабеле два разъёма и на одном разъёме устройство должно быть master, а на втором — slave. И никак иначе. Кто на каком? Неважно, главное — чтобы у обеих устройств статус был разный (см. ниже таблицу 2). Если интерфейс UDMA-66, 100 или 133, то кабель выглядит так: цветной разъём (розовый, синий, жёлтый, серобуромалиноый или «булыжного» цвета) — в системную плату, средний разъём — серый(!) для устройства slave, последний (крайний) разъём — чёрный(!) для устройства master. На обеих (на обеих!) устройствах, подключаемых на один кабель, перемычки ставятся в положение CS — cabel select, то есть, кто есть кто — определяется кабелем, точнее тем, на какой разъём вы подключите устройство.

Именование. В Windows это не столь важно, хотя значение имеет. В linux это имеет большое значение и выглядит так - см. таблицу 2.

Таблица 2. Статусы и имена устройств.

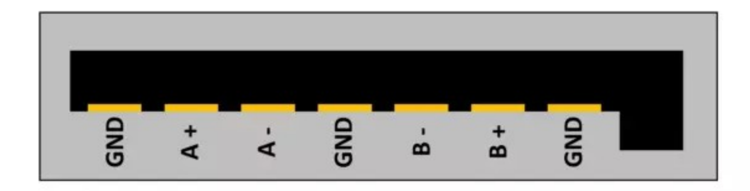
Статус устройства	Имя устройства
master-1	sda

slave-1	sdb
master-2	sdc
slave-2	sdd

То есть,, как мы подключили устройство - так мы его и именуем. А поскольку имя устройства — элемент полного пути к файлу, то, следовательно, для linux порядок подключения устройств имеет большое значение. С одной стороны ручное определение статусов — лишняя морока. Но с другой — точно знаешь, на каком hdd лежит файл. Если hdd много. Если один, то как подключили — значения не имеет, лишь бы статус был определён. Хотя правильно — если винт один, то он должен иметь статус master-1. Так вычислительной системе проще разбираться, где там винт подключен. Не следует грузить систему лишней работой по перебору каналов контроллера, она, ведь, и сбойнуть может, чисто случайно, а совсем не преднамеренно.

На Винде тоже в полный путь к файлу входит имя диска. Но что такое имя диска в Винде? А вот и нет! То, что вы подумали — это не имя диска. Но это типичные проблемы «системы для блондинок». То есть, на Винде те же проблемы с именованием, что имеются в linux, но ещё сложнее.

2) **SATA**. Здесь всё просто; ошибиться можно, только если сдуру разъём кверху ногами воткнуть, но для этого придётся приложить усилия и может, всё таки, дойдёт, что что-то идёт не так. Однако, авторы видели результаты подобных усилий. И всё-таки подобный «усиленный» случай — экзотический.



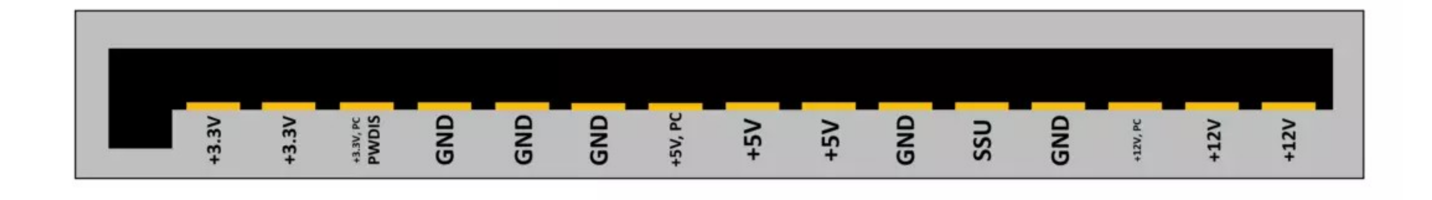


Рис. 9. Разъёмы SATA.

На рисунке 9 представлены разъёмы SATA: верхний — интерфейсный, нижний — питания.

Интерфейс. Для передачи и приема данных используется так называемый

дифференциальный сигнал: контакты A+ и A — для передачи инструкций и данных на жесткий диск, а контакты B для приема этих сигналов. Использование таких парных проводов значительно снижает влияние электрических помех в сигнале, следовательно, пропускная способность повышается.

Питание. Существуют по паре контактов каждого напряжения (+3,3, +5 и +12 В). Но большинство из них не используются, поскольку жестким дискам не требуется много энергии. Эта конкретная модель от Seagate (всё та же 3 Тб) потребляет менее 10 Вт при больших нагрузках. Контакты питания, помеченные как PC, являются "предварительно заряженными": эта функция позволяет вытаскивать и подключать жёсткий диск, пока компьютер продолжает работать (hot swapping — горячая замена).

PWDIS контакт позволяет сбросить жесткий диск удаленно, правда только с SATA версии 3.3, так что в этой модели винта (SATA 2.0) это просто еще одна линия +3,3 В. И последний контакт SSU лишь сообщает компьютеру, поддерживает ли винт технологию последовательной раскрутки шпинделей staggered spin up.

Это означает следующее. Блины внутри устройства должны раскрутиться на полную скорость, прежде чем компьютер сможет использовать их. Но если в ЭВМ установлено много жестких дисков (сервер, например), то внезапный и одновременный запрос в электроэнергии может нарушить работу системы. Технология постепенной раскрутки шпинделей помогает предотвратить возникновение таких проблем, но создаёт дополнительно несколько секунд задержки перед использованием винтов.

Определение статусов для дисков SATA. Поскольку на каждом кабеле один винт, то он всегда мастер.

Именование. А, вот, с именованием проблемы. Дело в том, что производители системных плат и программеры BIOS-ов при переходе на SATA создали нам, пользователям, проблемы. Как выше сказано, на интерфейсах IDE именование детерминировано: как подключил винт, так он и будет именоваться (см. таблицу 2). А с интерфейсами SATA такой предопределённости нет. А колодок SATA на современных системных платах от 4 и больше. И они, в отличие от колодок IDE, даже не обозначены на системной плате, которая из них первая, которая вторая и т. д. Более того, сами винты начинают идентифицировать себя по мере прохождения тестов самотестирования. Иначе говоря, при данном включении компа первым винтом (первым мастером) оказался вон тот винт (и он получит имя sda, а в Винде диск C будет вон на том винте) , а при следующем включении компа первым может оказаться вот этот винт (и тогда этот винт получит имя sda, а в Винде диск C будет на этом винте)). Но при подобном безобразии система работать не сможет, Это, ведь, понятно? Чтобы эту проблему решить — снова вернуть детерминированность, пришлось дорабатывать операционные системы: в linux появился новый модуль, определяющий имена устройств через унифицированные идентификаторы устройств (UUID), вычисляемые как

хэш-функции устройств по специальному алгоритму. Аналогично и в Винде. То есть, одни поленились и создали бяку, а другие потом героически преодолевали трудности, чтобы пользователь ничего не заметил.

Тем не менее, проблема была решена, хотя не так красиво, как было на интерфейсе IDE (см. рис. 10).

```
Файл  Правка  Вид  Поиск  Терминал  Помощь
/etc/fstab                                                           593/593           100%
proc             /proc                    proc     nosuid,noexec,gid=proc           0 0
devpts           /dev/pts                 devpts   nosuid,noexec,gid=tty,mode=620   0 0
tmpfs            /tmp                     tmpfs    nosuid                           0 0
UUID=ea5ecca2-713f-4538-9568-b6e2d391f86a          /        ext4     relatime         1        1
UUID=a736653e-0232-4931-9d59-f4009c17a982          swap     swap     defaults         0        0
# UUID=e50c7320-41b8-4cf1-a7bc-00fc9b62c370        /mnt/sdb1         ext3 defaults 0 0
# UUID=e4dbef5e-5f99-4baa-a0a3-f3ea213ebb22        /mnt/sda5         ext4 defaults 0 0
# UUID=b488498f-113f-44f1-97b2-df1c04b38aea        /mnt/sda1         ext4 defaults 0 0
# UUID=2019-10-23-07-36-57-00    /media/ALTLinux udf,iso9660      ro,noauto,user,utf8,nofail,
comment=x-gvfs-show  0 0
```

Рис. 10. UUID при определении устройств.

1.2.7. Особенности эксплуатации

1. Температурный режим. Очень важный момент, ибо винт — это точная механика и, следовательно, чувствителен к температуре. В таблице 3 приведена зависимость срока эксплуатации винта от температуры работы.

Таблица 3. Температура — наработка на отказ.

Температура, градусов С	**Коэффициент учащения отказов**	**Температурный коэффициент снижения времени наработки на отказ**	**Скорректи-рованное время наработки на отказ**
25	1,0000	1,00	232 140
26	1,0507	0,95	220 533
30	1,2763	0,78	181 069
34	1,5425	0,65	150 891
38	1,8552	0,54	125 356
42	2,2208	0,45	104 463
46	2,6465	0,38	88 123
50	3,1401	0,32	74 284
54	3,7103	0,27	62 678
58	4,3664	0,23	53 392

62	5,1186	0,20	46 428
66	5,9779	0,17	39 464
70	6,9562	0,14	32 500

Ориентировочно:

232140 часов / 24 часа = 9672.5 сут = 26.5 лет в среднем до выхода из строя.

32500 часов / 24 часа = 1354 сут., то есть, менее 4 лет в среднем до выхода из строя.

Правда, здесь имеется в виду непрерывная работа винта: включили — и он работает, работает, пока не выйдет из строя. У вас, конечно, не так.

Как определить температуру винта. Кладёте пальцы на винт:

- если чувствуете, что винт тёплый, то температура 40-45 градусов,

- если винт горячий, но вы эту температуру выдерживаете — 50-55 градусов.

- если винт горячий и через несколько секунд пальцы приходится убирать с винта — 60-65 градусов или выше; определённо, что-то надо делать, например, поставить дополнительную вентиляцию.

Либо, для определения температуры винта, можно воспользоваться соответствующей прогой, которая может сообщить температуру винта, а вы можете ей поверить.

2. Положение в пространстве. Самые популярные положения – горизонтальное, вертикальное на ребре и вертикальное на торце. Существует еще масса промежуточных положений под разными углами, однако на практике винчестер закрепляется в классических корпусах, имеющих стандартную форму и строго вертикальные/горизонтальные стенки (без наклонных поверхностей).

Однако, во-первых, никаких упоминаний о положении жесткого диска ни в инструкции по установке, ни в гарантийке нет. И это при том, что производители ревностно относятся к соблюдению правил эксплуатации своих изделий – сопроводительная литература пестрит напоминаниями, что неправильное использование лишает гарантии. Так что, если бы любое положение кроме горизонтального плохо сказывалось на сроке службы, об этом бы непременно написали на первых страницах макулатуры, поставляемой в комплекте с железками.

Во-вторых, обратите внимание на устройства хранения данных, в которых используются жесткие диски (бесчисленные NAS, внешние контейнеры и медиаплееры). Поверьте, производители этого железа, среди которых немало именитых компаний с отличным имиджем, изучили этот вопрос гораздо глубже нас - потребителей, они провели неоднократные консультации с производителями винчестеров относительно долговечности такой эксплуатации.

Даже сами производители винчестеров позволяют себе выпускать изделия с «нестандартным» расположением накопителей. Пример: внешний винчестер WD My Book Live Duo, положение двух винтов — вертикальное на торце. Уж кто-кто, а производитель винчестеров должен знать, как их правильно эксплуатировать. Если была бы хоть одна причина, не

позволяющая такое использование жесткого диска, вряд ли такое устройство увидело бы свет.

Очевидно, нет никаких причин считать, что вертикальное положение диска вредно и негативно влияет на продолжительность срока службы. Скорее всего, такой стереотип сложился в далекие времена, когда горизонтальное положение было необходимостью ввиду несовершенства технологий, и беспокойство специалистов по этому вопросу было небеспочвенным.

А вот чего действительно не рекомендуется делать, так это менять положение винчестера в пространстве во время работы. Это может привести к самым разным последствиям, но итог один – выход из строя и потеря информации. По этой же причине лучше не двигать и не переворачивать системный блок во время работы.

А если всё-таки вы заметили разницу в работе винта при разных положениях — смените блок питания.

1.3. Особенности флэшек и SSD

Концепция флэш-памяти была предложена компанией Toshiba в 1980 году, а первый коммерческий накопитель с флэш-памятью, твердотельный накопитель (SSD), был выпущен компанией SunDisk (позже переименованной в SanDisk) в 1991 году.

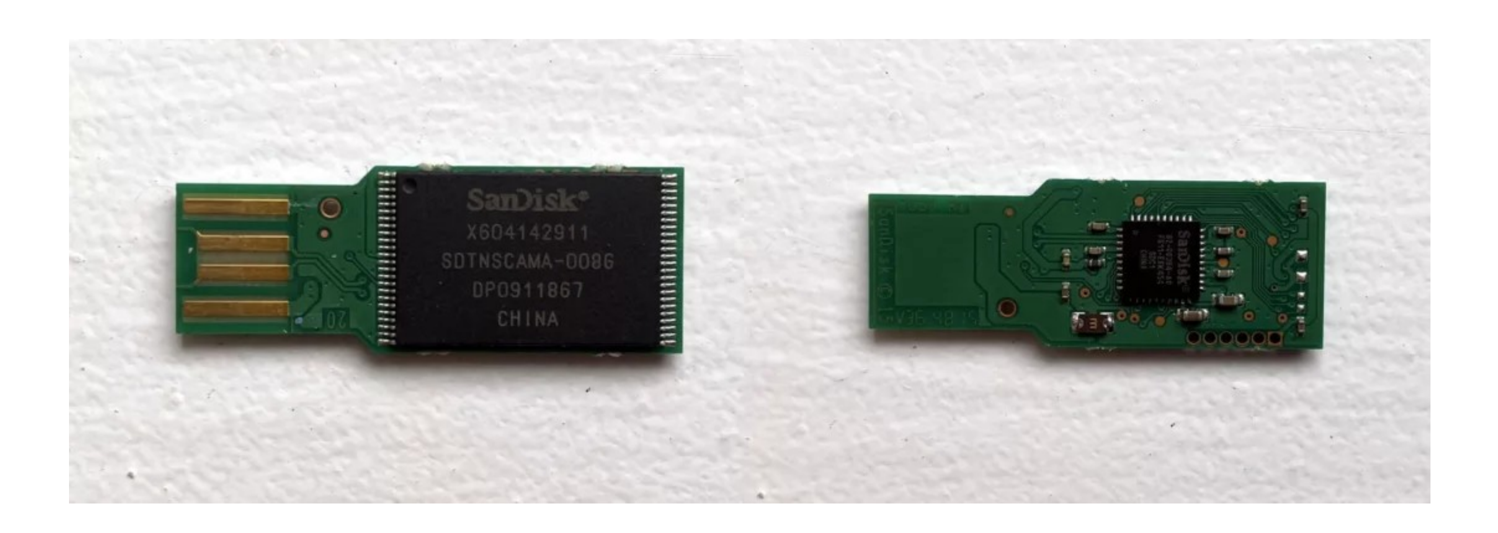


Рис. 11. Флэш-память («флэшка») без корпуса.

Слева на рисунке 11 на лицевой стороне флэшки находится микросхема NAND-памяти. Она заполнен миллионами «ячеек» сделанных из модифицированных транзисторов с плавающим затвором. В них используется высокое напряжение для записи и стирания заряда в ячейках и пониженное напряжение для считывания ячейки.

Если ячейка не заряжена, то при подаче этого низкого напряжения протекает ток. Это означает, что ячейка имеет состояние 0, и наоборот, если ячейка заряжена - ток не течёт, то состояние ячейки 1. Поэтому чтение из NAND flash происходит очень быстро. Однако, процессы

записи и удаления не столь быстродействующи.

Лучшие ячейки флэш-памяти — одноуровневые ячейки (SLC), обладают лишь одним уровнем заряда, применяемого на участке транзистора. Но существуют ячейки памяти с несколькими уровнями заряда. Они известны как многоуровневые ячейки (NAND MLC), и обозначаются как ячейки памяти с 4 уровнями заряда. Существуют ещё более сложные типы: трехуровневый (TLC) и четырехуровневый (QLC) имеют 8 и 16 различных уровней заряда соответственно.

Это влияет на то, сколько данных может храниться в каждой ячейке:

- SLC - 1 уровень = 1 бит
- MLC - 4 уровня = 2 бита
- TLC - 8 уровней = 3 бита
- QLC - 16 уровней = 4 бита

И так далее. На первый взгляд кажется, что QLC — самое лучшее. К сожалению, это не так. Поскольку потоки электрического тока очень малы и чувствительны к электрическому шуму, то для определения и подтверждения состояния заряда, его необходимо считывать несколько раз. Короче говоря, SLC — самый быстрые, но занимают наибольшее количество физического пространства (самые объёмные); QLC — самый медленные, но . . . больше битов за единицу деньгИ.

В NAND-памяти при отключении питания заряд сохраняется и утечка происходит очень медленно. К сожалению, использование высокого напряжения при записи повреждает ячейки, поэтому твердотельные накопители со временем изнашиваются. Для борьбы с этим используются процедуры для минимизации скорости износа — равномерное использование всех ячеек. Эта функция исполняется управляющей микросхемой (справа на рисунке 11, на обратной стороне флэшки),

Тесты показывают, что скорость чтения флэшек в два-три раза выше скорости чтения винчестеров, а запись в два-три раза медленнее. В реальности мы высокую скорость чтения не наблюдаем. Но это из-за того, что флэшки используют соединение по стандарту USB 2.0 с максимальной скоростью передачи данных всего 60 Мб/с, в лучшем случае USB 3.0 со скоростью передачи до 150 Мб/с, а жесткий диск с портом SATA 3.3 обладает пропускной способностью в несколько раз больше.

Для SSD ситуация лучше: флэш-память, используемая в лучших SSD сегодня — SLC или MLC, они работают немного быстрее и изнашиваются медленнее, кроме того, имеют объёмную структуру ячеек, что также повышает скорость. Также в них применяется интерфейс SATA 3.0, а иногда используется более быстрая система PCI Express через интерфейс NVMe. И в таком варианте скорость чтения и записи для SSD уже превышает возможности винчестеров в несколько раз.

Пример. SSD Samsung 850 Pro показан на рисунке 12.



Рис. 12. Samsung 850 Pro

На рисунке 13 показан он со снятой крышкой.

Рис. 13. Всё тот же Samsung 850 Pro

И сама печатная плата на рис. 14.



Рис. 14. Печатная плата.

На плате мы видим следующее:

- маленькие черные микросхемы — регуляторы напряжения,
- Samsung S4LN045X01-8030 — микросхема в центре: 3-ядерный процессор на базе ARM Cortex R4, который обрабатывает инструкции, данные, исправление ошибок, шифрование и управление износом,
- Samsung K4P4G324EQ-FGC 2 — микросхема вверху: 512 Мб памяти DDR2 SDRAM, используемой для кэширования,
- Samsung K9PRGY8S7M — микросхемы внизу и справа: каждая микросхема представляет собой 64 Гб 32-слойной вертикальной флэш-памяти NAND типа MLC (суммарно 4 микросхемы, две находятся на другой стороне платы).

Таким образом, в этом SSD используются 2-битные ячейки флэш-памяти, несколько

микросхем памяти и большая микросхема кэша, что приводит к повышению производительности. Поскольку запись данных во флэш-память происходит довольно медленно, то наличие четырёх микросхем флэш и большая память для для буферов позволяют выполнять запись параллельно и, соответственно, резко ускорить операцию записи.

Однако, несмотря на столь значительную конструктивную разницу между винчестерами, флэшками и SSD - эти устройства блоковые.

Следовательно, «флэшка» - это USB-устройство, в котором для хранения информации используется многоуровневая память типа NAND. Кстати, флэшки, выпускавшиеся 15-20 лет назад использовали одноуровневую NAND SLC и они, как правило, работают до сих пор, только ёмкость их измеряется всего лишь в сотнях мегабайт. Микросхема управления на них (контроллер флэшки) с внешней стороны (со стороны подключения в компу) реализует интерфейс USB-винчестера. Причём, поскольку изображается винчестер, то информационный обмен с компом идёт блоками по 512 байт (типа секторами). А с внутренней стороны контроллер флэшки управляет записью/чтением в NAND-память. При этом, место под запись очередного блока определяется по некоторому алгоритму так, чтобы равномерно задействовать всё пространство ячеек NAND-памяти. Также устроена и работает флэш-память в фотоаппаратах, плейерах, SD-картах, мобильных телефонах и прочих аналогичных устройствах.

Следовательно, SSD — это устройство в котором для хранения информации используется одно- или двухуровневая память типа NAND. Микросхема управления на них (контроллер SSD) с внешней стороны (со стороны подключения в компу) реализует интерфейс ATA-винчестера (интерфейс SATA). А с внутренней стороны контроллер SSD управляет записью/чтением в NAND-память почти как в обычных флэшках, но предпринимаются дополнительные меры по увеличению скорости записи/чтения и увеличению количества циклов записи.

И кстати, если в вашем ноутбуке только SSD, смело можете записываться в восхождение на Эльбрус. И даже не сомневайтесь — «звоните прямо сейчас». Ноутбук не забудьте.

1.4. Особенности CD/DVDROM

Патент на использование света и его отражении для хранения цифровых данных получил американский физик Джеймс Рассел в 1970 году. Что лежало в основе его идей (предистории его «идеи») - неизвестно. Компании Sony и Phillips приобрели лицензии и, отработав технологию, создали в 1978 году LaserDisc, а в 1982 году Compact Disc, ныне известный как CD. В 1987 году был создан ReWriteCD, перезаписываемый. В середине 90-х появились DVD, в 2003 году Blu-ray Disc (BD).

Сами диски (основа) изготавливаются из оргстекла с акриловым покрытием толщиной 0.6

мм. На нижнюю сторону основы наносится очень тонкий слой металла с хорошей отражающей способностью — серебро, золото или что-то аналогичное. На верхнюю сторону — материал, способный изменять фазу света при нагреве. Температуру создаёт луч маленького лазера. Запись информации производится непрерывной спиральной дорожкой, начиная с окружности недалеко от центра диска. При увеличении мощности лазера нагрев увеличивается и в оргстекле появляется углубление — пит (pit). Таким же образом происходит и чтение, только мощность лазера намного меньше, так чтобы фазовращающий материал не нагрелся. В зависимости от того, куда попал луч лазера — в углубление или в промежуток между углублениями, изменяется количество отражённого света, тем самым определяется 0 или 1 были записаны.

Чтобы на диск поместилось больше информации (CD –> DVD –> Blu-ray) для увеличения плотности записи используются лазеры с разной длиной волны (см. рис. 15).

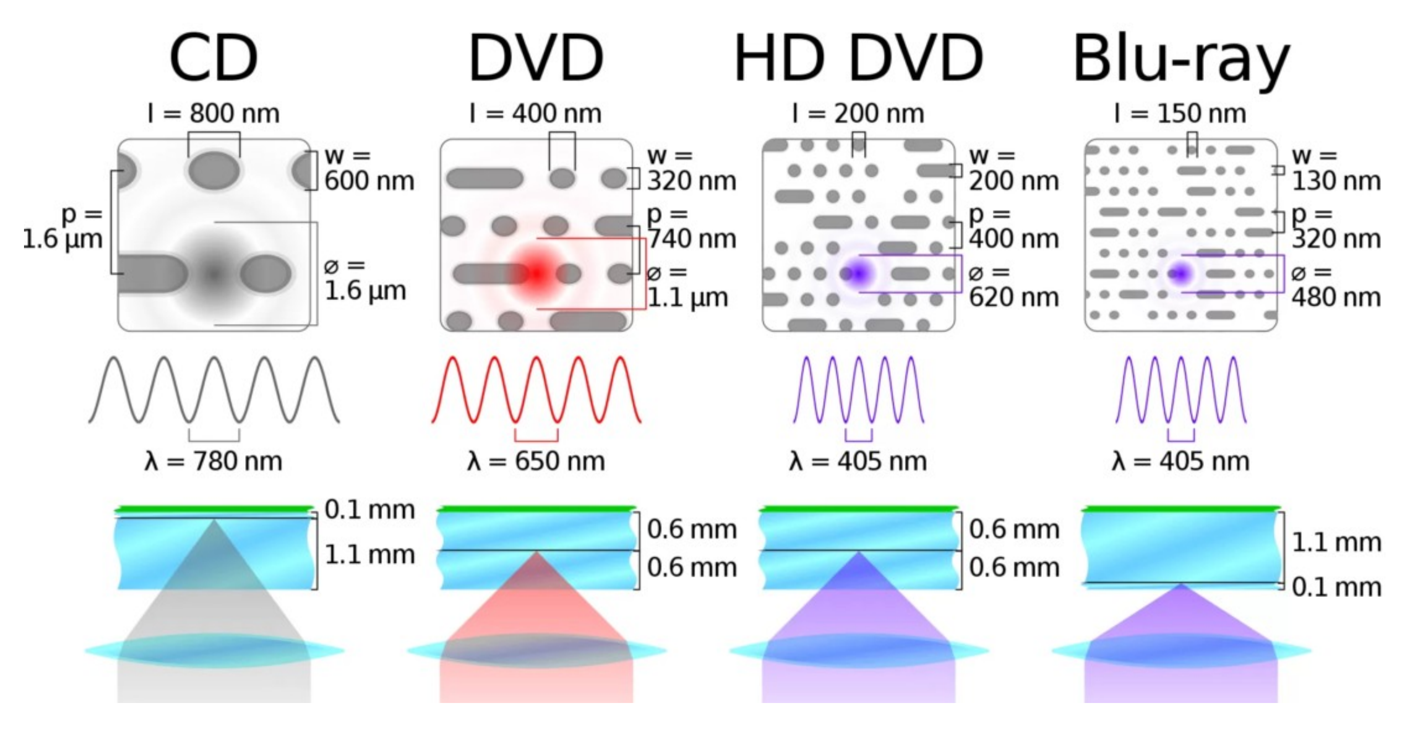


Рис. 15. Плотность записи и частоты лазеров.

Для CD используется инфракрасный лазер, DVD — красный, Blu-ray — фиолетовый. Таким образом, максимальный объём хранимой информации на CD — 0.84 Гб, на DVD — 4.7 Гб, на BD — 100 Гб.

Пример. См. рисунки 16, 17, 18.



Рис. 16. DVDROM от ноутбука. Видны шпиндельный двигатель в центре и лазерная головка ниже.

Рис. 17. Тот же DVDROM с нижней стороны. Видна червячная передача, перемещающая лазерную головку.

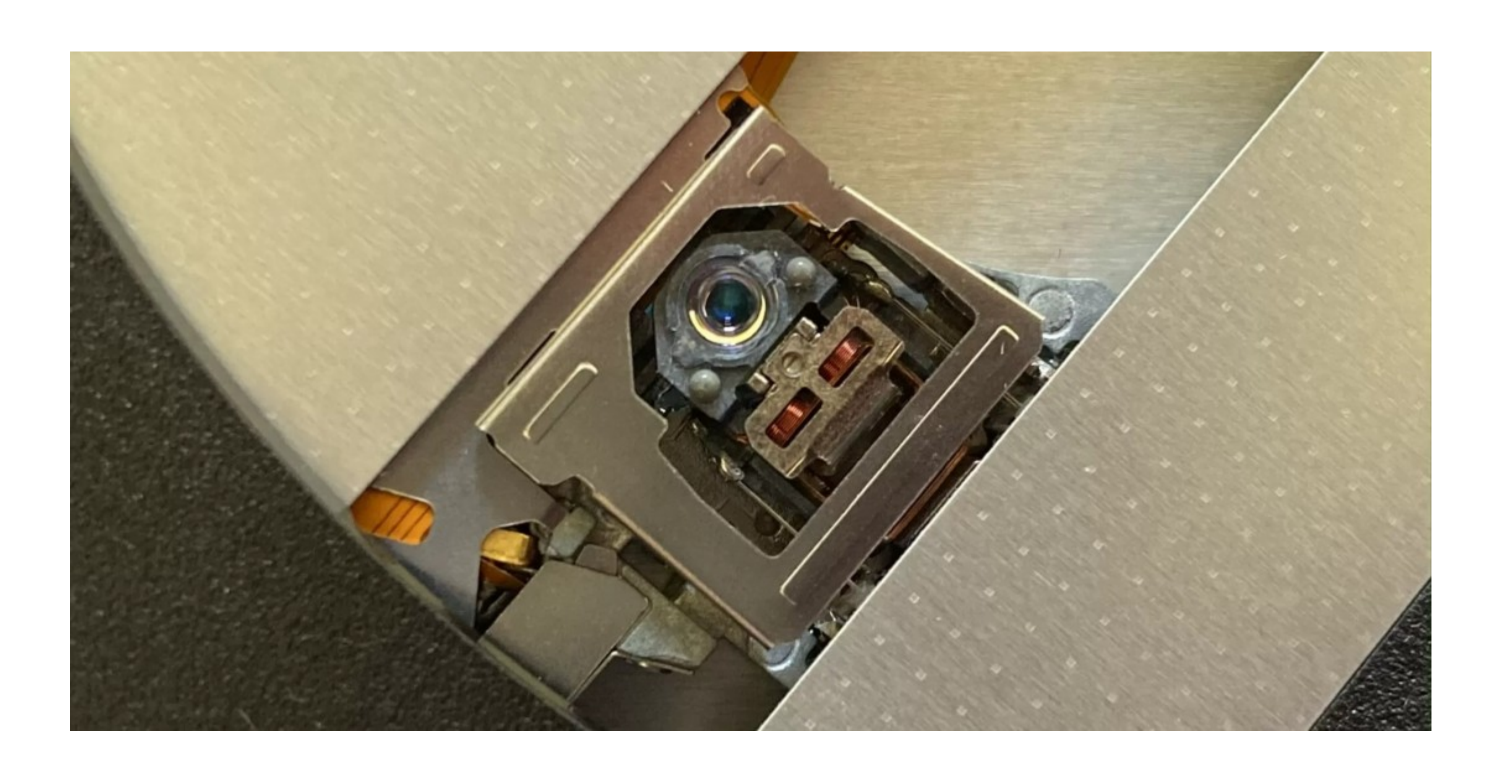


Рис. 18. Сама лазерная головка.

Достоинства оптических дисков.

1. Информация (диски) отделена от устройства.

2. Обычные (неперезаписываемые) диски нельзя отредактировать случайно или специально.

3. Срок хранения — почти «вечно» в прохладном тёмном месте, что идеально для архивов.

4. Дешевизна носителя (дисков).

Однако, тем не менее, данный вид устройств выходит из моды. Более предпочтительны оказываются флэшки (за счёт меньшего размера и большей скорости работы) или даже потоковые сервисы Интернета (за счёт доступности, «нематериальности» и опять же большей скорости работы). И куда девать библиотеку из сотен CD/DVD дисков с фильмами и литературой?

Но с другой стороны, на флэшках хранить информацию не очень надёжно, а облачные сервисы сегодня ещё есть, а завтра уже нет.

Вопросы на «засыпку»

1. Что такое низкоуровневое форматирование?
2. Есть ли на новом винчестере bad-блоки?
3. Как влияет загрузка информации из глобальной сети на срок эксплуатации винчестера?
4. Почему все советуют использовать SSD вместо HDD?
5. Что такое «выравнивания износа» на SSD?
6. Какие операционные системы поддерживают SSD?
7.

ЛЕКЦИЯ 2. ФОРМАТЫ РАЗБИЕНИЯ

2.1. Как видит ОС блоковые устройства

Во-первых, напомним определение операционной системы [1]:
Операционная система — это единственная (одна) программа, которая выполняется на вычислительной системе с момента включения (вычислительной системы) и до её выключения и которая выполняет следующие основные функции:
1) предоставляет процессам «абстрактную» вычислительную машину,
2) управляет ресурсами (доступом к ресурсам),
3) организует многозадачную среду и разграничивает процессы друг от друга,
3') организует многопользовательскую среду и разграничивает пользователей друг от друга,
4) учитывает процессы, ресурсы и пользователей.

Пункт 2) определения: «управляет ресурсами (доступом к ресурсам)» как раз относится к предмету лекции.

Блоковые устройства, в том числе и винчестер — ресурсы.

Но, чтобы чем-то управлять, оно, это управляемое, должно быть доступно. То есть, нужно иметь возможность оказывать воздействие на управляемый ресурс и иметь обратную связь с этим ресурсом, чтобы можно было оценить результаты воздействия и скорректировать при необходимости управляющее воздействие.

Иначе говоря, ОС должна «увидеть» устройство: опознать, определить марку/модель, определить его функциональность, характеристики и подобрать к нему программное обеспечение (прежде всего, драйвер). И только после этого предложить прикладному ПО доступ к нему и возможность его использования в определённых (контролируемых операционной системой) рамках.

Итак, вопрос: как ОС видит винчестер?

Чтобы ответить на этот вопрос, ещё раз вернёмся к определению: «**Операционная система — это . . . программа».**

Но программы пишут программеры.

Следовательно, ответ: ОС видит винчестер так, как это запрограммировал(и) программер(ы), когда он(и) разрабатывал(и) ОС.

Но, ведь, операционные системы разрабатывал не один и тот же программер или коллектив программеров. У каждой ОС свои разработчики/программеры. Крайне редко бывают случаи, чтобы один и тот же программер (или группа программеров) принимали бы участие в создании нескольких ОС. На слуху за последние полвека, пожалуй, только два случая: авторы unix предварительно принимали участи в разработке multix (совместный проект консорциума фирм) и авторы Windows предварительно принимали участи в разработке ОС для процессора альфа (проект фирмы DEC). В обоих случаях предварительные (первоначальные) проекты не были завершены: в первом случае консорциум распался из-за разногласий, во-втором - исчезла фирма DEC. А-а, вспомнил, был третий случай: авторы BeOS предварительно принимали участие в разработке MacOS. Ну, и конечно, достаточно часто бывают случаи, когда разработчики просто увольняются и находят работу в другой организации, но это одиночки, которые всего лишь исполнители и концепцию ОС не определяют.

А поскольку к настоящему времени операционных систем разработано многие тысячи, то определённо можно утверждать: функциональность, алгоритмы, характеристики операционных систем определяют разные люди.

Далее, вспомним, что ещё лет сорок назад Н. Вирт определил: «алгоритмы + структуры данных = программы» [12]. Структуры данных как раз и определяют тот объект (ресурс), что операционная система должна «увидеть». А в самой операционной системе находятся алгоритмы, которые с этими структурами данных работают, «разглядывая» очередное блоковое устройство и обеспечивая с ним работу. Ибо сказано: «**управляет ресурсами (доступом к ресурсам)».**

И, обратите внимание, только так и никак иначе: структуры данных определяют объект, а в программе ОС есть алгоритм для работы с этим объектом.

То есть, винчестер должен быть в ОС некоторым образом описан в структурах данных, чтобы ОС могла его опознать, начать им управлять и предоставлять доступ к нему прикладному ПО. И ещё раз: операционные системы пишут разные люди.

Вывод: сколько операционных систем в мире, столько и описаний (разных) объекта под названием винчестер. И, кстати, столько же и алгоритмов работы с ним.

Этот важный вывод нужно осознать и понять.

Если вы блондинка и пришли в ВУЗ учиться на ИТ-шника только затем, чтобы получить «корочку», а там и трава не расти, всё равно по специальности работать не будете, то вы работаете на Винде, и только на Винде, и всегда на Винде и вообще ничего кроме Винды от слова совсем. И весь мир вы видите сквозь Виндовые очки и он совершенно однозначен, никаких вариантов.

Но если ваша ИТ специальность для вас не пустое слово, и вы реально ИТ-шник, то вы смотрите на мир сквозь разные ОС. И вот тут уже всё не так однозначно, мир оказывается сильно разный и возможны варианты. В частности, вдруг оказывается, что винчестер в одной ОС работает, а другая его признаёт (он же исправный), но работать не хочет с ним. Почему? Ведь, он же исправный, ОС его «видит», на нём ваши очень ценные файлы, а ОС пишет: «Диск не форматирован. Вам его отформатировать?» И вы в шоке! Как отформатировать? Она что, сдурела? На нём же ваши суперценные и неповторимые шедевры!. . .

Следующий далеко идущий вывод: очевидно, что и описания винчестеров, и алгоритмы работы с ними в разных ОС разные. И почти всегда несовместимые друг с другом.

Известна задача: трём программерам дали одно и то же ТЗ. Рассадили их в разные места, чтобы они не общались друг с другом. Программеры запрограммировали. И хотя получившееся программы имели практически одинаковую функциональность (ТЗ то одно), но программы были разные.

А в случае с операционными системами и «Технические задания» — разные. Поэтому, так много операционных систем в мире, несовместимых друг с другом.

Итого, общий вывод: каждая ОС видит винчестер по своему, и работает с ним по своему. И если винчестер «подготовлен» в одной ОС, то другая, как правило, заявит, что та ОС «готовить винты» не умеет и это делать надо по другому.

2.2. Что же видит ОС в винчестере

Итак, как описан винчестер в ОС?

Чисто физические параметры винта операционную систему не интересуют. Тем более что, они у всех винтов одинаковы. Питание винтов SATA показано на рисунке 9. Питание винтов PATA почти ничем не отличается: +5 вольт для питания контроллера и +12 вольт для двигателей. Более того, и информационный интерфейс операционную систему не интересует, поскольку ОС работает с винтом только так: команда на запись блока + блок данных, отдать, всё; команда на чтение, ждать, получить блок данных, всё. Подробности этих команд реализуются контроллером ATA на системной плате, которому драйвер ОС эти команды передаёт. Команды типа «открыть файл», «закрыть файл», установить на позицию в файле (seek) — вообще до винта не доходят, это реализуется внутри ОС, на буферной памяти.

Тогда что же интересует операционную систему?

Как правило, при создании программ разрабатываются структуры данных и алгоритм совместно, методом неподвижной точки, то есть итеративно постепенно приближаясь к правильному решению, к «правильной» программе.

В ОС для эффективного управления устройством потребуется решать три основных группы задач:

- преобразования имён,
- расшаривания устройства,
- защиты данных и надёжности хранения.

Однако, алгоритмы преобразования имён и расшаривания устройства работают достаточно высоко, на уровне файловой системы.

А, вот, алгоритмы защиты данных и надёжности хранения по крайней мере в некоторой своей части работают на нижнем уровне, на уровне непосредственной работы с устройством.

Что должна знать ОС о винчестере?

Прежде всего, объём устройства, чтобы знать сколько файлов и какого размера можно сохранить на винте.

С точки зрения расшаривания устройства, защиты данных и надёжности хранения крайне желательно разделить винт на части («разделы») и в каждой части («разделе») хранить некоторую группу файлов, сходных по назначению и использованию. Например, системные файлы отдельно от пользовательских. Потому что, например, если система по некоторой причине «сдохнет», то возникает большая вероятность, что крах не затронет пользовательские файлы. Да и восстановить систему будет проще. Кроме того, файлы можно группировать по критерию: «изменяемые» (для чтения и записи; например, протоколы работы системы или файлы, создаваемые и редактируемые пользователем и др.) и «только для чтения» (например, программы, конфигурационные файлы и др.). Соответственно, «изменяемые» помещать в одну часть («раздел») винта, к которой будет полный доступ, а «только для чтения» - в отдельную часть («раздел») винта, к которой будет доступ только по чтению.

Таким образом, принимая во внимание только означенные требования, можно диск разбить на следующие части:

- раздел для системных программ (загрузчик, сама программа ОС, системные программы, их конфигурационные файлы — это всё однажды устанавливается, настраивается и больше не трогается до переустановки или перенастройки системы); режим доступа - «только для чтения»;

- раздел для изменяемых системных файлов (протоколы работы системы, временные файлы системных программ, архивы системных программ и пакетов и др.); режим доступа - «для чтения и записи»;

- раздел для пользовательских программ (прикладное ПО, то, что обеспечивает содержание операционной среды [1] пользователя; оно может изменяться при смене (обновлении) версий программ или при перенастройке операционной среды, что происходит достаточно часто); режим доступа - «для чтения и записи»; к сожалению, не получается данному разделу назначить режим доступа - «только для чтения» в силу его частой изменяемости;

- раздел для обычных пользовательских файлов и всего прочего; режим доступа - «для чтения и записи».

То есть, набирается четыре раздела диска даже для обычной пользовательской вычислительной установки (ПЭВМ). Иногда, на серверах, устраивают и ещё более детальное разбиение диска на части или даже используют соответствующее количество отдельных винчестеров для каждого назначения. Причём, перечисленные преимущества разбиения диска на разделы и хранения данных в разных разделах — это ещё не все плюсы подобного решения.

К сожалению, на практике редко (даже очень редко) подобное разделение диска реализуется. Почему? Потому что, это требует достаточной грамотности пользователя, ведь админы имеются не только лишь для всех пользователей, только некоторые пользователи могут себе позволить иметь админов. Это либо корпоративные пользователи, либо совсем уж «блондинки», а на всех админов не напасёшься.

Поэтому достаточно часто можно встретить использование ПЭВМ/ноутбука с одним разделом на винте.

Таким образом, если диск разбит на разделы, то ОС должна содержать некоторую структуру данных, типа таблица, в которой должно быть описано:

- количество разделов на винте (явно или косвенно; то есть, либо в этой структуре данных явно должно быть указано, сколько разделов на винте, либо ОС должна иметь возможность косвенно выяснить количество разделов, например, пересчитав их);

- для каждого раздела должно быть указано: адрес блока/сектора начала раздела, адрес блока/сектора конца раздела, количество блоков/секторов в разделе;

- для каждого раздела должно быть указано: некоторые характеристики/атрибуты раздела (тип раздела, назначение раздела, режим доступа к разделу и др.);

- для каждого раздела могут быть указаны: символьное имя раздела (это исключительно для пользователя), цифровое имя раздела (для ОС), метка раздела (метка это не имя, это метка);

- возможны некоторые дополнительные характеристики разделов.

И как уже было сказано в предыдущем пункте 2.1.1, формат этой таблицы является специфическим для каждой ОС.

Отсюда и следует вывод, что диск, определённый (разбитый на разделы) в одной ОС, не будет опознан в другой ОС. Ведь, структуры данных типа «таблица» - жёстко определяемые структуры данных.

Формат таблицы, описывающей разбиение винчестера на разделы, называется «форматом разбиения винчестера».

Очень часто эта таблица называется таблицей разделов (Partition Table, PT).

Иногда называется «метка диска» - disklabel.

Наиболее часто встречаемые форматы разбиения:

- PC BIOS, он же «msdos», он же «MBR» - используется в операционных системах OS/2, MSDOS, Windows; по известным причинам, это самый распространённый формат разбиения;
- gpt — новый самый распространённый формат разбиения, на который мы постепенно переходим, в связи с большими недостатками и ограниченностью формата PC BIOS; является основой для нового метода загрузки EFI;
- sun — используется в ОС SunOS, Solaris;
- bsd - используется в ОС FreeBSD, OpenBSD, NetBSD, DragonFlyBSD и родственных;
- aix - используется в ОС AIX (unix компании IBM);
- pc98 — разрабатывался для проекта ПЭВМ в Японии, но проект «не пошёл», а формат в памяти народной остался;
- amigo — когда-то выпускалась ПЭВМ Amigo, на которой было много игрушек, ПЭВМ уже давно нет, а формат в памяти народной остался;
- sgi — используется в ОС IRIX (фирмы SGI);
- apple (или mac) - использовалось в ОС MacOS (версий до 8-ой, то есть, до перехода на FreeBSD);
- а также: dvh, loop и многие-многие другие.

Из всего этого многообразия, вы наверняка столкнётесь с первыми двумя:

- PC BIOS (msdos) — потому, что почти во всех старых ПЭВМ винчестеры разбиты на разделы и описаны в этом формате, или, если ПЭВМ новая, но винчестер небольшой ёмкости,
- gpt — потому что переходим на этот новый формат.

2.3. Примеры форматов разбиения дисков

2.3.1. Формат PC BIOS

Пример разбиения винчестера на разделы в формате PC BIOS (он же формат msdos, он же формат MBR) показан на рис. 19, а первый сектор — на рис. 21.

Обратите внимание на следующие прискорбные особенности данного формата:

1) Таблица разделов (главная PT) — в единственном экземпляре. Потому что, тогда, давным давно, в 1982 году, когда этот формат впервые появился в ОС PC DOS 2.0 (разработка фирмы IBM), винчестеры были достаточно большие (типоразмер 8 дюймов), но по объёму были ещё маленькие — 10-30 Мегабайт всего лишь. Ну, что там делить-то? Всего то каких-то задрипанных 10 Мб. Поэтому очень часто на винте был только один раздел и только очень редкие пользователи-мазохисты-параноики (помешанные на надёжности) позволяли себе создавать больше одного раздела на винте. И всё равно, на 10-30 Мегабайтах не сильно размахнёшься с количеством разделов. То есть, процедура разбиения была неактуальной и разработчики формата посчитали достаточной и то, что таблица в единственном экземпляре, и то, что в таблице всего

лишь четыре строки. Более того, через несколько лет, когда к разработке ОС подключилась Microsoft (присвоив себе лавры), она вообще стала использовать почти всегда только первые две строки таблицы, посчитав остальные избыточными. Да-да, полное отсутствие «полёта фантазии» в предвидении будущего.

2) Строк в таблице только четыре. Обоснование — больше, дескать, нет места. И очень долго, до середины 90-х, этого действительно хватало. На серверах, если требовалось больше дисковой памяти, просто ставили дополнительные винты. И только в 90-е, когда появились винчестеры в сотни Мегабайт, вопрос встал и оказался актуальным. И увидели все, что четырёх строк мало. И придумали «костыль» - расширенный раздел (extended, тип 5), в котором стали создавать «логические разделы». И показалось народонаселению, что это хорошо. Потом, правда, при ближайшем рассмотрении и опыте эксплуатации, выяснилось, что не всё так радужно, и выявилась проблема: логические разделы (см. рис. 19, 20) описываются односвязной списковой структурой, а у неё есть неприятная особенность — при потере ссылки весь конец списка теряется. То есть, четыре первичных раздела описываются более-менее — не скажем, надёжно, но всё-таки, а то, что в extended совсем кое-как.

Вывод 1. При создании разделов нужно всегда по максимуму задействовать главную PT. И только если вам нужно создать 5, 6, . . . и более разделов, тогда создаём последним (**!последним**) в главной PT расширенный раздел и в нём располагаем эти сраные логические разделы.

Вывод 2. Расширенный раздел всегда создаётся последним в главной PT и ВСЕГДА «НА_ВСЁ_ОСТАВШЕЕСЯ_МЕСТО». Иначе у вас будут проблемы. Мы вас предупредили, не балуйтесь. Если не прислушаетесь, то вы можете оказаться в ситуации, когда свободные сектора (и даже много) на диске будут, но вы не сможете их использовать.

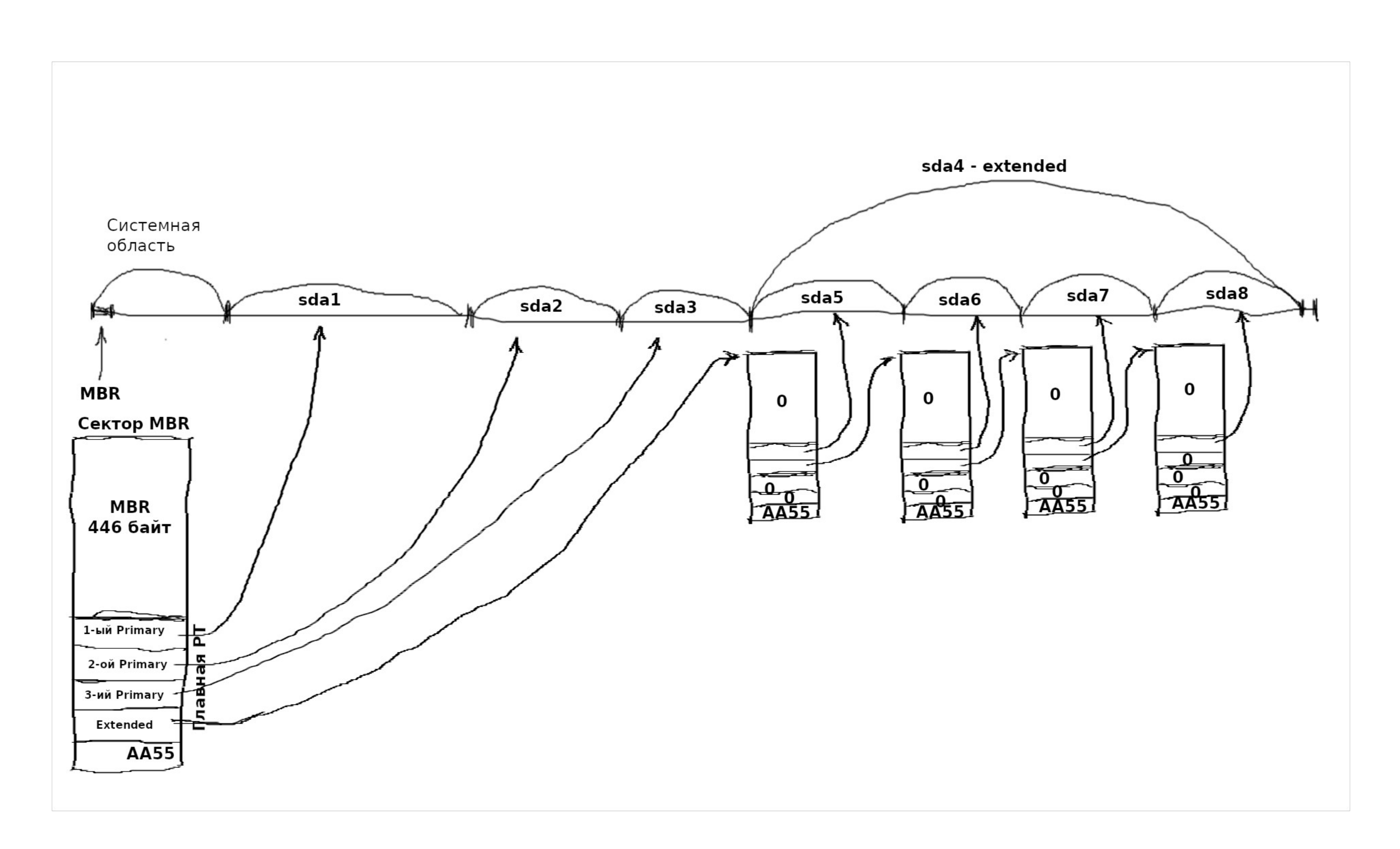


Рис. 19. Разбиение винта в формате PC BIOS.

3) И кроме того, выявилась проблема с именованием разделов. Суть проблемы в следующем. Имена разделов, описанных в главной PT — фиксированы и определяются номером строки: строка 3 — тогда раздел sda3 и никак иначе. А, вот, имена в разделе extended могут меняться. Предположим, что вам нужно по какой-то надуманной причине найти место, чтобы разместить «нечто_исключительно_важное» и вы не придумали ничего лучше, как удалить совсем раздел sda6 и на его месте создать новый раздел, куда и запихнуть это «нечто_исключительно_важное». При этих манипуляциях получится то, что показано на рисунке 20. То есть, раздел sda7 станет разделом sda6, раздел sda8 станет разделом sda7, а бывший раздел

sda6, пересозданный, получит имя sda8. Иначе говоря, при удалении элемента из середины списка, список «схлопывается», при этом имена элементов, расположенных после удалённого, становятся на единицу меньше, а конец списка оказывается на новом месте. А, ведь, эти имена используются в абсолютных путях файлов. И если некая ваша прога работала с данными, расположенными в /sda7/data, то теперь их там не будет.

Вывод: удалять разделы можно только с конца, с последнего. Иначе будут проблемы.

Кстати, в Винде ситуация аналогичная, буквы дисков также меняются, если не заблокировано автоматическое переназначение букв дискам.

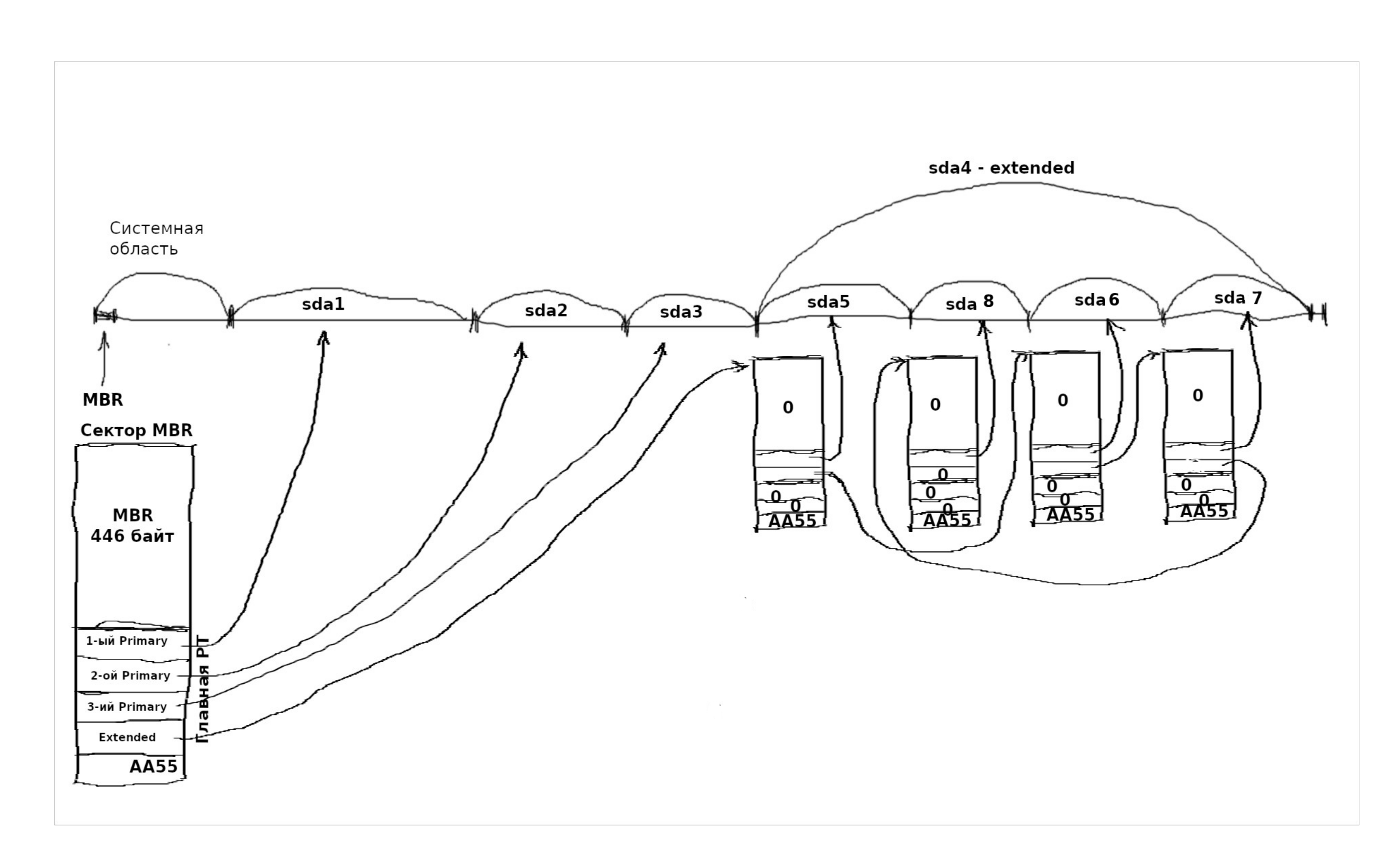


Рис. 20. Изменение имён при пересоздании разделов.

4) Обратите внимание на маленький «недовес» в конце дискового пространства. Эта «усушка, утряска», в размере от нескольких сотен до нескольких тысяч секторов (обычно, несколько Мегабайт) возникает в связи со спецификой физического разбиения диска на разделы — диск делится на разделы всегда по границам цилиндров — это контроллер так срабатывает: ведь, он же работает с блинами в адресации CHS. А как было сказано в первой лекции, в конце диска (внутренние дорожки) создаётся служебная область, размером в Мегабайты, десятки Мегабайт — зависит от производителя и модели винчестера. То есть, часть последнего цилиндра, а иногда и больше — заняты. Соответственно, конец видимого пространства может оказаться где-то «на цилиндре». Этот остаток, неполный цилиндр, и оказывается этой «усушкой, утряской». Не помогает даже использование адресации LBA и указание точного адреса последнего сектора последнего раздела, всё равно внутри-то используется геометрическая адресация.

Смещение	Длина, байт	Описание	
0000h	446	Код загрузчика	
01BEh	16	Раздел 1	*Таблица разделов*
01CEh	16	Раздел 2	
01DEh	16	Раздел 3	
01EEh	16	Раздел 4	
01FEh	2	Сигнатура (55h AAh)	

Рис. 21. Структура сектора MBR более подробно.

Разделы, описанные в четырёх строках главной PT, называются **первичными** разделами. Разделы, создаваемые в расширенном разделе — **логические** разделы. Каждый логический раздел описывается своей собственной PT. Поэтому, в этом формате разбиения таблиц разделов может быть много.

2.3.2. Формат gpt

В самом конце 2000 года фирмой Intel была выпущена спецификация EFI (Extensible Firmware Interface, «интерфейс расширяемой прошивки», с которой всё и началось. Эта спецификация представляла интерфейс между операционной системой и микропрограммами, управляющими низкоуровневыми функциями оборудования, его основное предназначение: корректно инициализировать оборудование при включении системы и передать управление загрузчику или непосредственно ядру операционной системы. Если кратко, то EFI предназначен был для замены BIOS.

EFI имеет поддержку формата GPT, который свободен от характерных для формата PC BIOS ограничений, но также поддерживает и старую схему разметки дисков PC BIOS.

По первоначальному предложению формат GPT должен был иметь следующую схему:

- в начале и в конце винчестера создаются системные области размером по 2 Мб, по 4 тысячи секторов каждая при секторе, равном 512 байт, или по 1 тысячи секторов, при секторе равном 4 Кб. В каждой из этих системных областей создаются по две PT, растущих навстречу друг другу. То есть, в начальной системной области первая PT (основная, главная) идёт от начала (от 2-ого сектора) к середине, а вторая от конца системной области к середине, навстречу друг другу. Аналогично, в концевой системной области также две PT заполняются навстречу друг другу. Длина строки PT — 512 байт. Иначе говоря, должно было реализовываться четырёхкратное резервирование такой важной для винта информации, как PT. В каждой таблице разделов должно было быть до 2 тысяч строк, то есть, предполагалось создавать до двух тысяч разделов.

Однако, Microsoft, как всегда побежала впереди паровоза и как всегда всё испохабила: в 2002 году в своей Windows-2000 реализовала только две таблицы разделов, одну в начале и одну в конце, посчитав, что и так сойдёт. Да и эти две таблицы реализовала в сильно урезанном виде. Остальным операционным системам (прежде всего, linux) пришлось подстраиваться под это решение в целях совместимости.

Поэтому в настоящее время формат GPT выглядит так, как показано на рис. 22 и 23. Для лучшего понимания авторы приводят два рисунка.

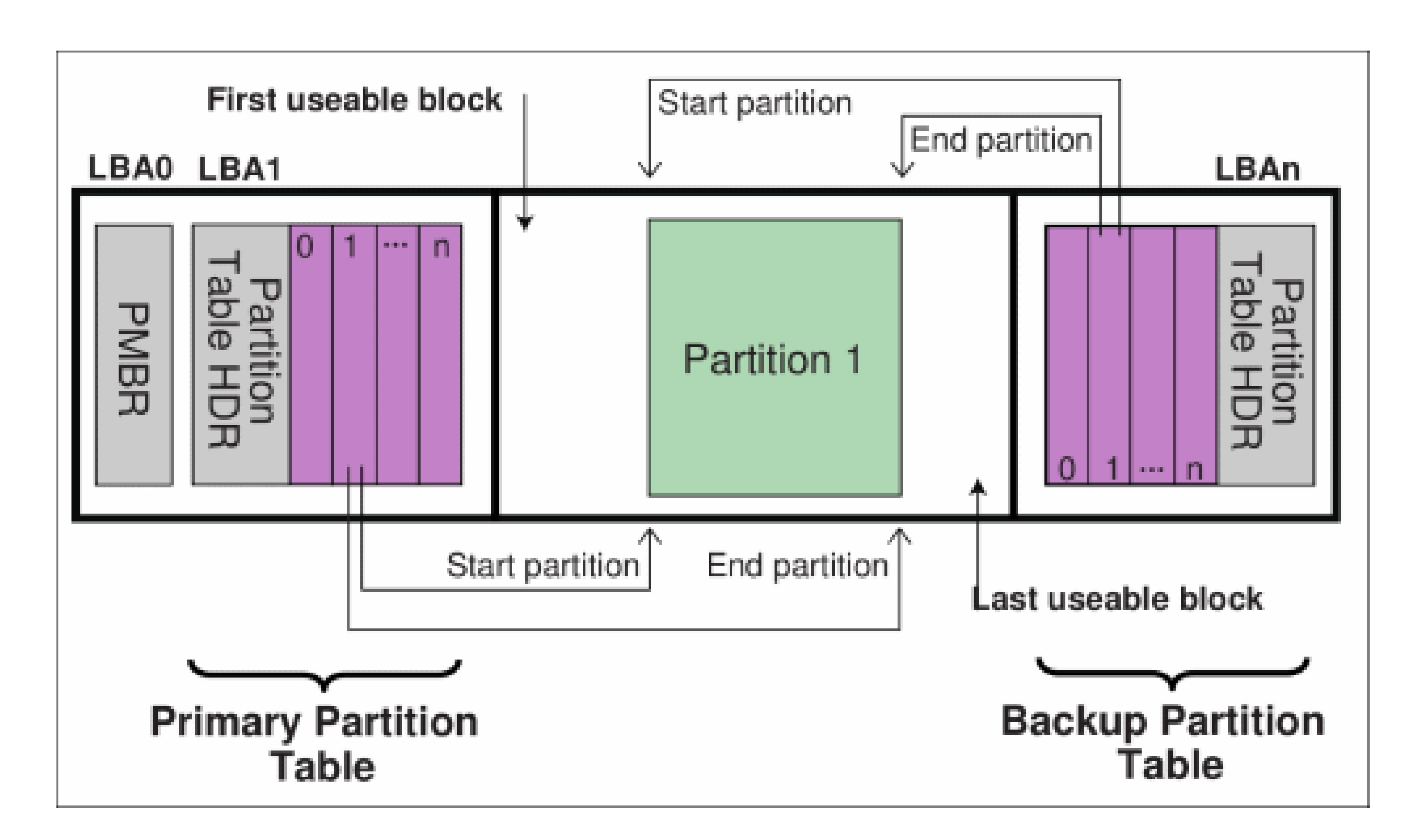


Рис. 22. Схема разбиения диска в соответствии с форматом GPT.

Таким образом, вся структура GPT на жестком диске состоит из 6 частей:

Таблица 4. Состав структуры GPT.

LBA-адрес	Размер (секторов)	Назначение
LBA 0	1	Защитный MBR-сектор
LBA 1	1	Первичный GPT-заголовок
LBA 2	32	Таблица разделов диска
LBA 34	NN (сколько будет)	Содержимое разделов диска
LBA -34	32	Копия таблицы разделов диска
LBA -2	1	Копия GPT-заголовка

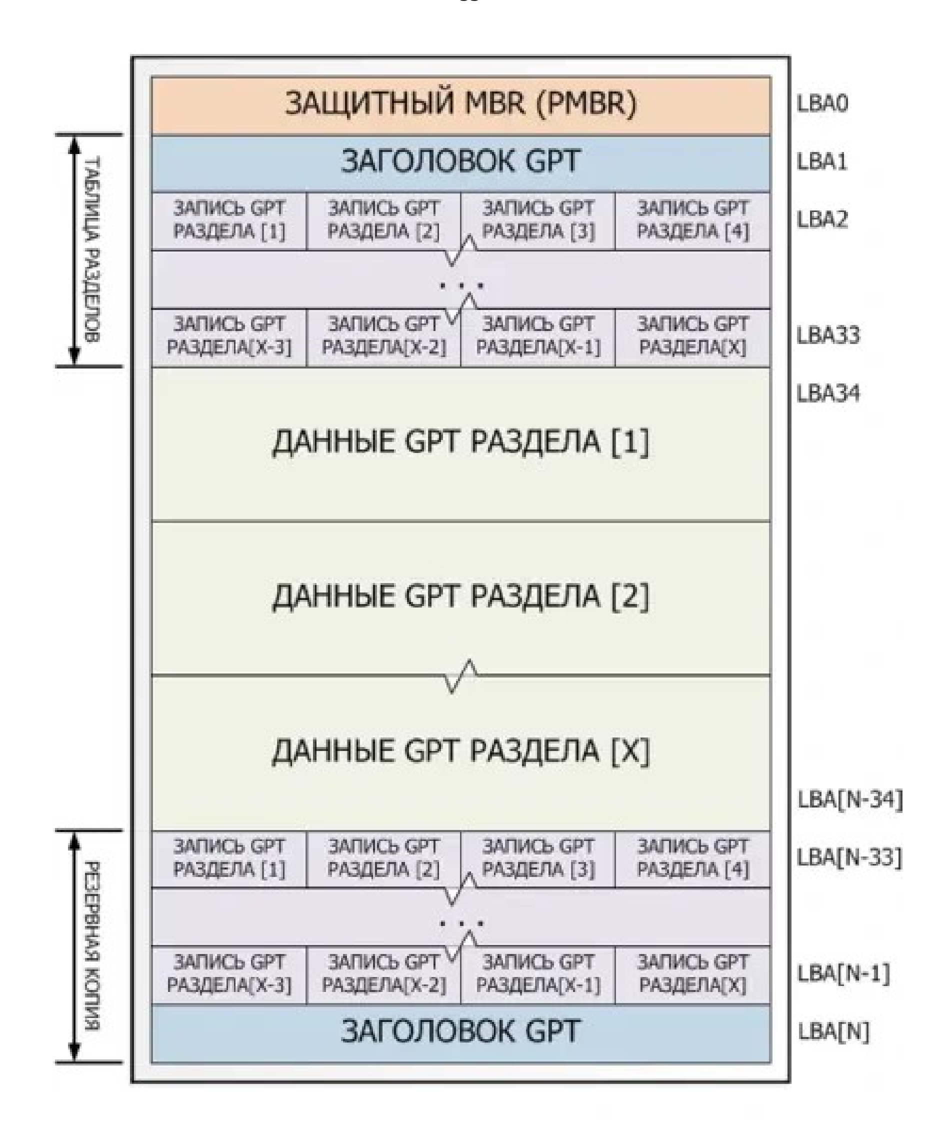


Рис. 23. Схема разбиения диска в соответствии с форматом GPT — альтернативный взгляд.

1. Защитный MBR-сектор

Первый сектор на диске в формате разбиения gpt (с адресом LBA 0) — это все тот же MBR-сектор (показан на рис. 21). Он оставлен для совместимости со старым программным обеспечением и предназначен для защиты GPT-структуры от случайных повреждений при работе программ, которым про GPT ничего не известно. Для таких программ структура разделов будет выглядеть как один раздел, занимающий все место на жестком диске.

Структура этого сектора ничем не отличается от обычного сектора MBR. Но в его таблице разделов должна быть создана единственная запись с типом раздела 0xEE. И этот раздел должен начинаться с адреса LBA 1 и иметь размер 0xFFFFFFFF. В полях для CHS-адресации раздел соответственно должен начинаться с адреса 0/0/2 (сектор 1 занят под саму MBR) и иметь конечный CHS-адрес FF/FF/FF. Признак активного раздела должен иметь значение 0 (неактивный).

При работе компьютера с UEFI, данный MBR-сектор просто игнорируется и никакой код в нем также не выполняется.

2. Первичный GPT-заголовок

Этот заголовочный сектор (таблица 5) содержит в себе данные о всех LBA-адресах, использующихся для разметки диска на разделы.

Таблица 5. Структура GPT-заголовка.

Смещение (байт)	Размер поля (байт)	Пример заполнения	Название и описание поля
0x00	8 байт	45 46 49 20 50 41 52 54	Сигнатура заголовка. Используется для идентификации всех EFI-совместимых GPT-заголовков. Должно содержать значение 45 46 49 20 50 41 52 54, что в виде текста расшифровывается как "EFI PART".
0x08	4 байта	00 00 01 00	Версия формата заголовка (не спецификации UEFI). Сейчас используется версия заголовка 1.0
0x0C	4 байта	5C 00 00 00	Размер заголовка GPT в байтах. Имеет значение 0x5C (92 байта)
0x10	4 байта	27 6D 9F C9	Контрольная сумма GPT-заголовка (по адресам от 0x00 до 0x5C). Алгоритм контрольной суммы — CRC32. При подсчёте контрольной суммы начальное значение этого поля принимается равным нулю.
0x14	4 байта	00 00 00 00	Зарезервировано. Должно иметь значение 0
0x18	8 байт	01 00 00 00 00 00 00 00	Адрес сектора, содержащего первичный GPT-заголовок. Всегда имеет значение LBA 1.
0x20	8 байт	37 C8 11 01 00 00 00 00	Адрес сектора, содержащего копию GPT-заголовка. Всегда имеет значение адреса последнего сектора на диске.
0x28	8 байт	22 00 00 00 00 00 00 00	Адрес сектора с которого начинаются разделы на диске. Иными словами — адрес первого раздела диска
0x30	8 байт	17 C8 11 01 00 00 00 00	Адрес последнего сектора диска, отведенного под разделы
0x38	16 байт	00 A2 DA 98 9F 79 C0 01 A1 F4 04 62 2F D5 EC 6D	GUID диска. Содержит уникальный идентификатор, выданный диску и GPT-заголовку при разметке
0x48	8 байт	02 00 00 00 00 00 00 00	Адрес начала таблицы разделов
0x50	4 байта	80 00 00 00	Максимальное число разделов, которое может содержать таблица
0x54	4 байта	80 00 00 00	Размер записи для раздела, пока 128 байт
0x58	4 байта	27 C3 F3 85	Контрольная сумма таблицы разделов. Алгоритм контрольной суммы — CRC32
0x5C	420 байт	0	Зарезервировано. Должно быть заполнено нулями

Система UEFI проверяет корректность GPT-заголовка, используя контрольные суммы, вычисляемые по алгоритму CRC32. Если первичный заголовок поврежден, то проверяется контрольная сумма копии заголовка. Если контрольная сумма копии заголовка правильная, то эта копия используется для восстановления информации в первичном заголовке. Восстановление также происходит и в обратную сторону — если первичный заголовок корректный, а копия неверна, то копия восстанавливается по данным из первичного заголовка. Если же обе копии заголовка повреждены, то диск становится недоступным для работы. Вот, он, результат «гадкости» MS. А, ведь, должно было состояться ещё две проверки на корректность.

У таблицы разделов дополнительно существует своя контрольная сумма, которая записывается в заголовке по смещению 0x58. При изменении данных в таблице разделов, эта

сумма рассчитывается заново и обновляется в первичном заголовке и в его копии, а затем рассчитывается и обновляется контрольная сумма самих GPT-заголовков.

3. Таблица разделов диска

Следующей частью структуры GPT является собственно таблица разделов (таблицы 6, 7). В настоящее время операционные системы Windows и Linux используют одинаковый формат таблицы разделов — максимум 128 разделов (это результат самоуправства MS), на каждую запись раздела выделяется по 128 байт, соответственно вся таблица разделов займет 128*128=16384 байт, или 32 сектора диска.

Таблица 6. Формат записи раздела.

Смещение (байт)	Размер поля (байт)	Пример заполнения	Название и описание поля
0x00	16 байт	28 73 2A C1 1F F8 D2 11 BA 4B 00 A0 C9 3E C9 3B	GUID типа раздела. В примере приведен тип раздела "EFI System partition". Список всех типов можно посмотреть здесь
0x10	16 байт	C0 94 77 FC 43 86 C0 01 92 E0 3C 77 2E 43 AC 40	Уникальный GUID раздела. Генерируется при создании раздела
0x20	8 байт	3F 00 00 00 00 00 00 00	Начальный LBA-адрес раздела
0x28	8 байт	CC 2F 03 00 00 00 00 00	Последний LBA-адрес раздела
0x30	8 байт	00 00 00 00 00 00 00 00	Атрибуты раздела в виде битовой маски
0x38	72 байта	EFI system partition	Название раздела. Unicode-строка длиной 36-символов

Таблица 7. Значения битов атрибутов раздела.

Номер бита	Назначение бита
бит 0	Указывает необходимость раздела для функционирования системы. OEM-разработчики могут таким образом защищать свои разделы от перезаписи дисковыми утилитами
бит 1	прошивка UEFI не должна читать отсюда (не привязывать EFI_BLOCK_IO_PROTOCOL)
бит 2	бит 2 - загрузочный раздел для BIOS (менеджер загрузки UEFI должен игнорировать этот раздел)
биты 48-59	Для частного использования
бит 60	Помечает раздел как доступный только для чтения. Используется только для "Microsoft Basic Data Partition" с типом {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}
бит 61	Shadow copy, Microsoft
бит 62	Помечает раздел как скрытый. Используется только для "Microsoft Basic Data Partition" с типом {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}
бит 63	Предотвращает автоматическое назначение буквы диска данному разделу. Используется только для "Microsoft Basic Data Partition" с типом {EBD0A0A2-B9E5-4433-87C0-68B6B72699C7}

С оставшимися частями разметки все понятно и без подробного описания. Содержимое

разделов — говорит само за себя. Копия таблицы разделов — тоже понятно, хранит копию таблицы разделов. Ну и последний сектор диска — это копия GPT-заголовка.

При загрузке Linux из BIOS при использовании GPT возникает проблема размещения «полуторного» загрузчика (core.img), который при использовании MBR хранился в системной области между MBR и первым разделом. Для её решения создаётся вспомогательный раздел biosboot размером 1 Мб, в который и записывается программа «полуторного» загрузчика, которая понимает формат GPT и может быть запущена с помощью первичного загрузчика (boot.img) из защитной MBR. Раздел biosboot содержит бинарный код размером около 30 КБ без файловой системы.

4. EFI System Partition

EFI System Partition (ESP, /boot/efi после загрузки) - раздел MBR или GPT, размером 50-200 Мб. Используется UEFI для загрузки ОС, содержит файловую систему FAT32, FAT16 или FAT12 (точнее собственный стандарт на основе подмножества FAT), в которой хранятся загрузчики, образы ядра и драйверы, а также утилиты и журналы. Первый сектор раздела содержит загрузчик, который может быть использован BIOS или UEFI в режиме CSM (Compatibility Support Module). Некоторые реализации UEFI самостоятельно переключаются в режим CSM встретив ESP раздел для MBR. Загрузчики GRUB2 и elilo будучи загруженными UEFI с раздела ESP продолжают загрузку ядра самостоятельно с поддерживаемых ими устройств и файловых систем.

Синтаксис имён файлов в этом разделе соответствует FAT (8.3, ASCII, всегда прописные) с расширением длинных имён (LFN, ASCII или UCS-2, но нет поддержки UTF-16), то есть, каталог отделяется символом "\", регистр букв не принимается во внимание ('A' = 'a'). Выделенные имена файлов: ".", ".." (точка и две точки).

Атрибуты файлов: только чтение, скрытый, системный, каталог, архив.

ESP содержит каталог /EFI, в котором размещаются каталоги BOOT (загрузчик последней надежды bootx64.efi для архитектуры x86_64, при этом не должно быть других файлов для той же архитектуры) и каталоги загрузчиков (Microsoft/BOOT/bootmgfw.efi, redhat, ubuntu, centos и т.д.). Каталог /efi/boot/bootstr.nvr может содержать EFI переменные.

2.3.3. Формат bsd

Ещё один формат разбиения диска, с которым вы с большой вероятностью может встретиться — формат bsd. Он используется операционными системами *BSD, например, FreeBSD, DragonFlyBSD и др.

Для разделов в BSD-таблице предусмотрено восемь (FreeBSD) или шестнадцать (DragonFlyBSD) записей. Соответствующие им разделы маркируются литерами -- от a до h (или -- до p в случае с DragonFly). То есть, разделов, казалось бы, может быть создано 8 (или, соответственно, 16). Однако практически это не совсем так (вернее, совсем не так).

Дело в том, что одна из записей (третья по счету, маркируемая литерой c) резервируется для описания всего диска в целом. Далее, первая запись таблицы (литера a), отводится для описания корневого раздела файловой системы. А очевидно, что на конкретной локальной машине корневой раздел данной ОС может быть только один, вне зависимости от количества дисковых разделов и даже физических дисков.

Наконец, вторая запись, маркируемая литерой b, предназначена исключительно для описания раздела подкачки (swap-раздела), который, во-первых, не может содержать данные, и во-вторых, является единственным на весь диск (хотя при наличии двух физических дисков поделить между ними пространство подкачки -- идея вполне здоровая).

```
# /dev/ad0s1:
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
  a:   524288       63    4.2BSD     2048 16384 32776
  c: 16771797       63    unused        0     0
  # "raw" part, don't edit
  d:   524288   524351    4.2BSD     2048 16384 32776
  e:   524288  1048639    4.2BSD     2048 16384 32776
  f:  1048576  1572927    4.2BSD     2048 16384     8
  g: 14150357  2621503    4.2BSD     2048 16384 28552
```

Рис. 24. Так выглядит PT, если предварительно используется формат PC BIOS.

Если же весь наличествующий диск планируется отдать на растерзание какой-либо BSD-системе, то проще создать один-единственный слайс на (почти) весь его объем, оставив записи в BIOS-таблице для остальных неиспользованными. Ну, а семи (и, тем более, 15) позиций BSD-таблицы обычно достаточно для обособления всех необходимых ветвей файловой системы. Впрочем, в ряде случаев целесообразно и разнесение разделов по двум слайсам, но -- не более.

Разметка диска, использующая записи в PT сектора MBR, называется разметкой в режиме совместимости (см. рис. 24). Вне зависимости от того, создаётся ли один слайс для BSD-системы или несколько отдельных -- для каждой операционки, в режиме совместимости в начале диска резервируется системная область в размере 63 секторов (всего около 30 Кбайт), в котором не только сохраняется в неприкосновенности стандартный MBR, но и остаётся место для записи кода какого-либо стороннего загрузчика. В итоге диск будет доступным для других операционных систем, по крайней мере теоретически и Винда не скажет, что неформатированный. Хотя разделы bsd всё равно не опознает.

Однако использование режима совместимости и PT сектора MBR во FreeBSD не является обязательным. Вполне допустимо записать в сектор MBR непосредственно BSD-таблицу разделов. В этом случае понятно, что слайсов как таковых не возникнет, а все дисковое пространство будет представляться как один слайс и может быть разбито на BSD-партиции по тем же правилам, что и слайс отдельный. И тут становится ясной необходимость резервирования третьего поля BSD-таблицы -- именно в ней и описывается весь диск, целиком отведённый под BSD-систему (см. рис. 25).

$ bsdlabel /dev/ad0

```
# /dev/ad0:
8 partitions:
#        size   offset    fstype  [fsize bsize bps/cpg]
  a:   524288        0    4.2BSD    2048 16384 32776
  b:  2074624   524288      swap
  c: 156301488       0    unused    0     0     0
  # "raw" part, don't edit
  d:   524288  2598912    4.2BSD    0     0     0
  e: 10240000  3123200    4.2BSD    0     0     0
  f: 142938288 13363200   4.2BSD    0     0     0
```

Рис. 25. Так выглядит PT, если используется весь диск.

Такое обращение с диском именуется режимом эксклюзивного использования, или Dangerously Dedicated. Вопреки названию, в нем не таится никакой опасности ни для данных пользователя, ни для его здоровья. А единственная подстерегающая его опасность -- это то, что диск в эксклюзивном режиме не будет опознан никакой другой операционной системой, установленной на данном компьютере (обращению к диску по сети он препятствий не составит). Однако это -- чисто теоретическое неудобство, потому что ни одна из распространённых

операционных систем все равно не умеет толком работать с BSD-разделами и файловой системой FreeBSD (особенно современной — UFS2). Даже у linux есть до сих пор проблемы. А, скажем, при наличии на другом физическом диске мультизагрузчика GRUB, FreeBSD с "эксклюзивного" диска вполне может быть им загружена.

В документации по FreeBSD встречаются указания, что "эксклюзивные" диски иногда не могут быть загрузочными, вероятно, потому, что BIOS не сможет опознать нестандартные записи в MBR. Однако, видимо, это относится к каким-либо старым версиям BIOS.

Тем не менее, в документах проекта FreeBSD всегда подчёркивается, что эксклюзивный режим -- в частности, из-за грошовой экономии дискового пространства, -- следует использовать лишь в исключительных случаях. Один из резонов к такому использованию -- несоответствие "геометрии" диска, видимой из BIOS, и того представления о ней, которое складывается у FreeBSD.

Схема разметки диска в BSD-стиле принята и в других ОС этого семейства, с той разницей, что термин слайс ни в OpenBSD, ни в NetBSD не применяется. А в NetBSD, кроме литеры «c», предназначенной для описания всего диска, резервируется еще и литера «d» -- она также предназначается для описания первичного раздела целиком.

В ОС BSD-семейства может быть иным и формат таблицы разделов слайса. В частности, в DragonFlyBSD формат позволяет разделить слайс не на 8, как во FreeBSD, а на 16 логических разделов. Впрочем, и в текущей версии FreeBSD предельное число разделов на один слайс увеличено, хотя отражения в официальной документации это пока не нашло. Да и штатными средствами её инсталлятора выделить более восьми разделов внутри слайса не получается.

Тем не менее, FreeBSD достаточно широко распространена у Российских провайдеров Интернета, поэтому встреча с ней вполне возможна. К тому же, она наличествует в Реестре допустимого для использования программного обеспечения для бюджетных организаций.

2.4. Загрузчики

Помимо таблиц разделов на винчестерах иногда появляются загрузчики. Они появляются в тех случаях, когда на винчестере где-то, в каких-то разделах, устанавливаются операционные системы. То, что на винчестерах могут быть загрузчики, подчёркивает особую роль винчестеров для вычислительных систем: винчестеры являются не только хранилищем файлов, но и местом, откуда периодически зарождается функциональность вычислительной системы, то есть, грузится операционная система и другое программное обеспечение, формирующее операционную среду для пользователя.

В зависимости от их назначения загрузчики делятся на первичные, полуторные, вторичные и, иногда, ещё и третичные.

2.4.1. Первичные загрузчики

Они же иногда называются досистемными загрузчиками, потому что, используются/работают тогда, когда на компьютере ещё вообще ничего нет, никакого программного обеспечения, ибо сразу после включения — память чистая.

На интеловской архитектуре после нажатия кнопки питания, по броску питания, автоматически запускается программа самотестирования системной платы. Если она завершилась без ошибок — все устройства на плате, включая процессор, функционируют нормально, - то управление передаётся программе BIOS (basic input-output system — базовая ввода/вывода система). Это программа состоит из множества модулей, занимающихся низкоуровневым (на уровне интерфейсов) взаимодействием с разнообразными устройствами ввода-вывода - как с теми,

на которых могут храниться программы (жесткие и гибкие диски, магнитные ленты и даже сетевые карты), так и теми, посредством которых можно общаться с пользователем (последовательные порты передачи данных - если есть возможность подключить консольный терминал, системная клавиатура и видеокарта - для простых персональных рабочих станций).

Модули программы BIOS начинают работу с оборудованием. В частности, один из модулей проверяет энергонезависимую память CMOS, содержащую некоторые параметры настройки системы. Это та самая память, данные из которой мы видим, когда «заходим в BIOS» и которые даже можем немного изменить и, тем самым, слегка изменить поведение вычислительной системы, причём, иногда так «слегка», что система перестанет работать.

После проверки оборудования и сбора информации от всех опознанных устройств вычислительной системы, программа BIOS решает, может ли вычислительная система продолжать работу. Если может, тогда наступает решающий момент: BIOS смотрит параметры в памяти CMOS и определяет, что грузить (какую программу) и откуда её грузить. Чаще всего в таблицах CMOS указано, что грузить нужно программу с винчестера. Как правило, он один. Иногда указывается устройство, имитирующее винчестер (флэшка, SSD).

Если указан винчестер, то BIOS читает первый сектор винчестера (MBR) и грузит его в оперативную память (см. рис. 26). Затем BIOS проверяет, что, собственно, оно загрузило: последние два байта загруженного сектора равны AA55 или нет. Если нет, то BIOS сообщает пользователю: «Не системный диск. Вставьте системный диск». И ждёт действий пользователя. Как правило, ожидается нажатие кнопок Reset или питания.

Если последние два байта загруженного сектора равны AA55, то BIOS считает, что загруженный сектор является программой «первичный загрузчик» (MBR - master boot record) и передаёт управление на первую команду этой программы.

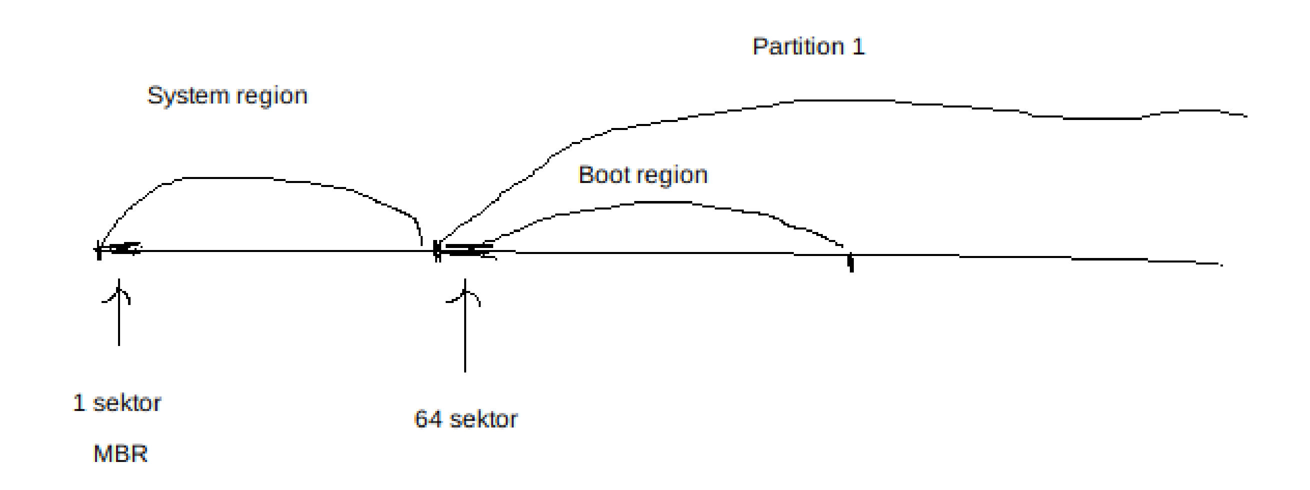


Рис. 26. Начало винчестера в формате разбиения PC BIOS.

Алгоритм первичного загрузчика простой: просмотреть таблицу разделов, которая находится тут же в этом же секторе и найти в ней строку, в первой графе которой находится значение 80h — признак активного раздела, то есть, раздела, на котором точно установлена какая-то ОС. Из загрузочной области этого раздела (из самых первых секторов) загрузить вторичный загрузчик и передать ему управление.

На этом этап досистемной загрузки заканчивается.

Однако так работает только простейший «старинный» вариант досистемной загрузки.

Если в вычислительной системе установлен загрузчик GRUB, то первичный загрузчик модифицирован им и грузит полуторный загрузчик GRUB`a, который обычно располагается в секторах со 2-го по 62 системной области винчестера.

Если вычислительная система достаточно новая с UEFI, то первичный загрузчик также модифицированный и грузит вторичный загрузчик из специального небольшого загрузочного

раздела, который является самым первым разделом на винчестере и располагается сразу после системной области.

На неинтеловской архитектуре вообще всё по-другому, но в данном пособии этот вопрос не рассматривается.

2.4.2. Другие (последующие) загрузчики

Другие загрузчики, которым передаётся управление после первичного, уже работают с файловыми системами и будут рассматриваться в последующих лекциях.

2.5. Средства разметки дисков

Как правило в составе системного программного обеспечения есть программа fdisk. Программа, предназначенная для работы с таблицами разделов и, тем самым, для разметки винчестера на разделы. Она входит в состав базового системного ПО для практически всех операционных систем, по крайней мере, для всех, широко распространённых.

Она даже может называться по-другому, но она таки есть. Просто потому, что, если ОС умеет работать с винчестерами, то ОС должна уметь их «готовить».

Очевидно, что в разных дистрибутивах эти fdisk`и разные. Разные и по функционалу и по интерфейсу. Пользователя обычно прежде всего интересует (и пугает) вопрос безопасности работы с ней, ведь, очень часто работать с таблицами разделов приходится не только на новых («чистых») дисках, но и на находящихся в работе.

И с этой точки зрения (безопасности для данных) наиболее удобной и, заодно, многофункциональной, является программа fdisk из дистрибутива linux. Она характеризуется тем, что при запуске считывает таблицы разделов в память и далее работает с этими образами в памяти. Таблицы разделов (реальные) на диске модифицируются только по специальной команде «write». Поэтому пользователь, без сомнения, может семь раз отмерить, на восьмой ещё раз всё проверить на предмет ошибки и . . . отказаться вносить изменения, дав команду q (quit - «выход без сохранения»).

Для формата gpt есть её модификация — gdisk, очень похожая по интерфейсу.

К сожалению, далеко не во всех дистрибутивах программа fdisk является столь же безопасной. Особенно не рекомендуется работать с fdisk из дистрибутивов Windows — она крайне опасна: пользователь даёт команду и программа тут же, сломя голову, бежит её выполнять. То есть, легко можно получить приключения, потеряв информацию на диске. С Виндовой fdisk можно работать только с «чистым» диском.

Для удобства пользователей созданы графические варианты программ разметки дисков, (например, gparted), которые красиво и достаточно понятно представляют структуру диска.

Но. Консольные программы fdisk присутствуют в системе практически всегда, ибо входят в базовый набор системного ПО. А, вот, графические нужно, как правило, дополнительно устанавливать.

2.6. Заключительные выводы по лекции

1. Винчестер — самая важная часть компьютера.

2. Винчестером управляет операционная система, поскольку это очень важный ресурс для неё и для всей вычислительной системы.

3. Даже в очень простой MS DOS самой сложной и содержательной частью был файл msdos.sys, содержавший алгоритмы и структуры работы с винчестером.

4. Каждая ОС «видит» винчестер по-своему. В основе этой «точки зрения» лежат свои, особенные для ОС, структуры данных , описывающих винчестер и свои особенные алгоритмы работы с винчестером.

5. Эта «особенность» точек зрения операционных систем на винчестеры порождает понятие «формат разбиения» винчестера на разделы.

6. И как следствие этого, винчестер, разбитый на разделы в одной ОС, как правило, не опознаётся в другой.

7. Потому что, при разбиения винчестера на разделы на винчестер пишутся специфические для ОС структуры данных, описывающих это разбиение. Поскольку в другой ОС эти структуры данных другие, то эта другая ОС не может работать с «чужим» винчестером.

8. Если на винчестер в какой-либо раздел устанавливается ОС, то в системную область винчестера уже в процессе установки пишется первичный загрузчик.

9. Для работы с таблицами разделов винчестеров в составе дистрибутивов наличествует специальная программа для этого — fdisk.

Вопросы на «засыпку»

1. Что будет с винчестером, если все сектора пометить как bad?
2. Может ли каталог и файл в нём иметь одинаковое имя?
3. Является ли кластер конвертом для диска?
4. Можно ли выполнить низкоуровневое форматирование дискеты?
5. Можно ли с помощью программы mkfs скорректировать скорость вращения диска?
6.

ЛЕКЦИЯ 3. ФОРМАТИРОВАНИЕ

3.1. Процесс форматирования

Итак, диск подключен, разбит на разделы.

Следующая операция — создание файловых систем в разделах. Эта операция называется форматирование раздела. В процессе форматирования на раздел пишутся структуры данных файловой системы.

Обратите внимание: Данное форматирование — это не разбивка диска на дорожки-сектора. Разбивкой блинов на дорожки-сектора занимаются на заводе-производителе винчестеров на больших и дорогих станках. Серворайтеры называются. Один раз в жизни винта. Это, так называемое, низкоуровневое форматирование. А то действо, что делаем мы — высокоуровневое форматирование, можно делать сколько угодно раз, хоть каждый день (и даже без перерыва «на обед») и заключается оно в том, что на уже размеченные дорожки и в обозначенные метками сектора мы, с помощью специальных системных программ, записываем структуры данных файловой системы с помощью которых будем потом работать с файлами. Эти структуры данных файловой системы ничем особым от наших пользовательских данных не отличаются, это такие же таблицы или списковые структуры, которые мы можем создавать и в наших прикладных программах. Только используются эти структуры данных потом операционной системой для организации хранения наших файлов.

3.2. Файл и файловая система

Определение. Файл — это именованная область на диске, размером от 1-ого блока до некоторого предела, устанавливаемого файловой системой. Эта область может быть занята некоторыми данными, а может быть пустой — зарезервированная область под что-то. Эта именованная область может быть составной (фрагментированной), то есть, состоять из нескольких групп блоков, расположенных в разных местах диска (точнее, раздела).

Файл не может занимать место на диске меньше одного блока (сектора), хотя размер данных в файле может быть от одного байта до «очень_много». В файле могут быть любые данные, что может сгенерить вычислительная система.

Определение. Файловая система — это метод организации хранения файлов на разделе (на диске, на SD-карте, на диске SSD, в общем, на носителе). Это конструктивное, технологически правильное определение файловой системы.

Другие определения.

Определение-2: Файловая система — это подсистема ОС, управляющая ресурсом «дисковое пространство». Её цель и назначение - эффективное управление данным ресурсом (https://habr.com/ru/post/108629/). Такое целеполагание означает, что основным содержанием файловой системы являются алгоритмы. И это правильно.

Определение-3 из википедии: **Фа́йловая систе́ма** — порядок, **определяющий** способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании: цифровых фотоаппаратах, мобильных телефонах и т. п.

Термины «метод», «порядок», «способ» - подразумевают наличие алгоритма, реализующего этот «метод», «порядок», «способ» и некоторых данных, с помощью которых этот «метод», «порядок», «способ» реализуется. Естественно, это есть и в файловой системе. Современные файловые системы современных многозадачных (и даже многопользовательских) операционных систем включают два основных алгоритма:

- алгоритм преобразования имён: символьных «человеческих» в цифровые имена операционной системы и соответствующего устройства хранения информации,

- алгоритм «расшаривания» устройства (точнее, файловой системы (как объекта) на устройстве) между многочисленными программами, выполняющимися в системе, и пользователями, работающими в системе.

А есть ещё алгоритмы обеспечения надёжности и защиты, но о них мы в данном пособии говорить почти на будем, дабы не растекаться мыслью по древу, ибо сказано: «Не пытайся объять необъятного!» ((С)К.П.)

3.3. Реализация файловых систем

Эти алгоритмы в основном реализованы в коде самой операционной системы (иногда бывает и вне, но редко, например, в случае использования файловых систем пространства пользователя). Структура реализации файловых систем в ядре linux показана на рис. 27.

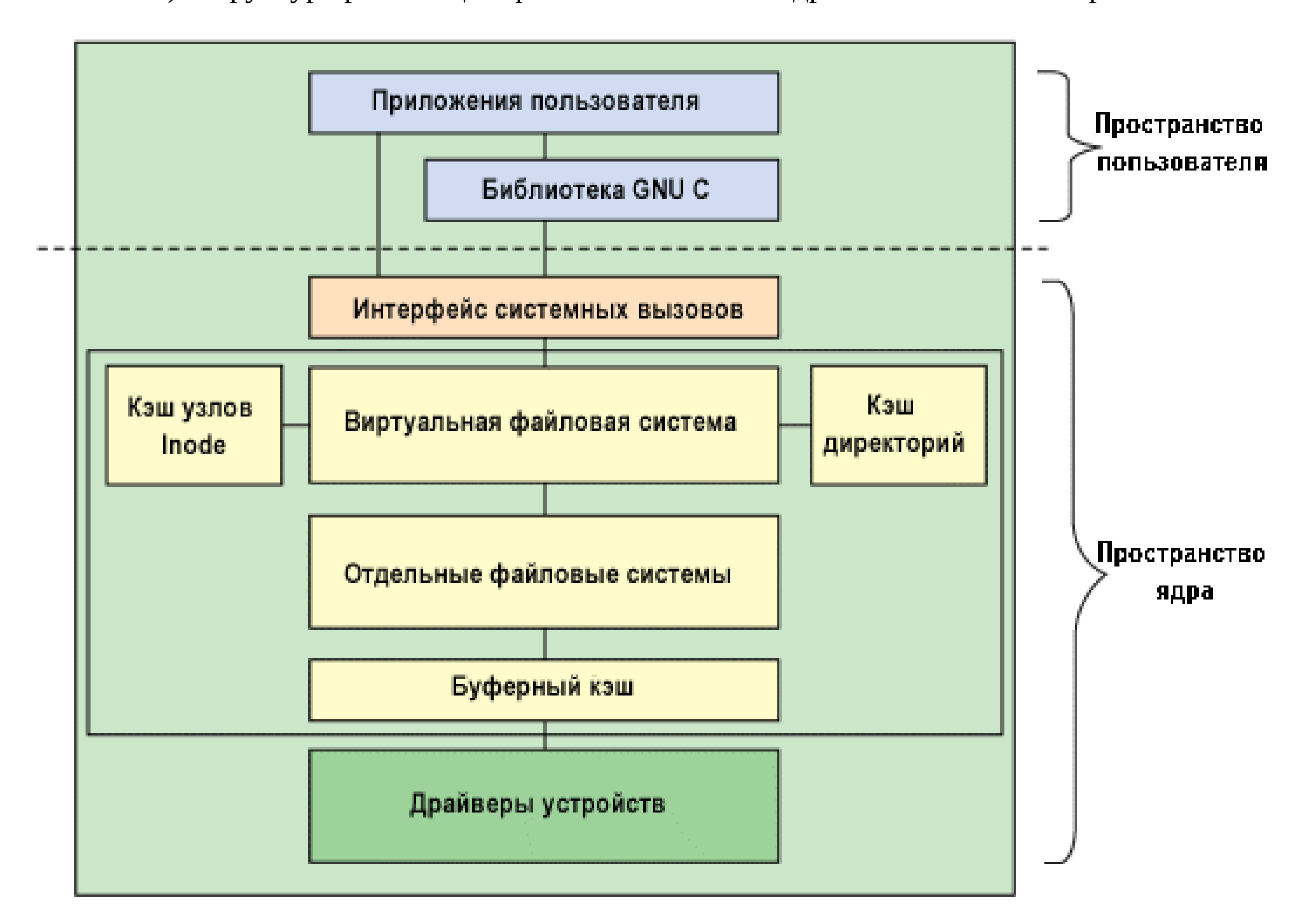


Рис. 27. Подсистема ввода/вывода linux: реализация файловых систем.

Для программ пользователя библиотека libc предоставляет интерфейс для доступа к системным вызовам (API ОС). Можно и напрямую работать (программить) с ситемными вызовами, но неудобно, сложные они.

Интерфейс системных вызовов (API) действует как коммутатор, направляющий системные вызовы из пространства пользователя в соответствующие точки пространства ядра, в том числе, к VFS — виртуальной файловой системе.

VFS является основным интерфейсом к файловым системам нижнего уровня. Этот компонент экспортирует набор интерфейсов файловых систем и передаёт вызовы в отдельные файловые системы, образ поведения которых может быть весьма различным. Для объектов файловой системы существуют два кэша: кэш узлов inode и кэш каталогов. Каждый из них предоставляет пул недавно использованных объектов файловых систем. Не данных из файлов, а

именно объектов файловых систем, тех самых таблиц и списковых структур, что были записаны в разделы при форматировании. А для кэширования данных есть отдельный «буферный кэш — он на рисунке 27 показан ниже. Зачем эти кэши, ведь они занимают весьма много памяти? А вдруг последует повторное обращение к тем же данным? Или к близко расположенным. Эти кэши — это олицетворение «принципа локальности», который обсуждался в [1] в главе об управлении памятью в ЭВМ. А также реализация альтернативы: либо память, либо скорость. В файловых системах упор сделан на скорость, в настоящее время для пользователей это более критично.

Каждая файловая система реализована специальным модулем операционной системы — драйвером файловой системы. Они обобщённо скрыты под прямоугольником «Отдельные файловые системы». Не путайте драйверы файловых систем с драйверами устройств. Драйверы файловых систем - это «логические» драйверы, чисто программные, они с аппаратурой не работают, их содержание — алгоритмы файловых систем. А драйверы устройств — это драйверы, реализующие аппаратные интерфейсы устройств, на рис. 27 они в самом низу. То есть, драйверы файловых систем работают поверх драйверов устройств.

Реализация (драйверы) каждой файловой системы, например, ext2, JFS, ntfs и так далее, экспортирует общий массив интерфейсов, который используется (и ожидается) виртуальной файловой системой (VFS). Буферный кэш буферизирует запросы между файловыми системами и блочными устройствами, которыми они могут управлять. Например, через буферный кэш проходят запросы на чтение и запись к драйверам устройств. Это позволяет кэшировать запросы для более быстрого доступа (вместо обращения непосредственно к физическому устройству). Буферный кэш управляется набором списков последних использованных элементов.

Обратите внимание, что командой sync можно сбросить буферный кэш на носитель (принудительно отправить все незаписанные данные на драйверы устройств и, в дальнейшем, на устройства хранения). Потому что, современные универсальные операционные системы работают в «асинхронном» режиме: то есть, при загрузке операционная система считывает структуры данных файловых систем в память и в дальнейшем при работе пользователя с файлами меняются именно эти структуры данных в памяти. И сами данные кэшируются — см. рисунок 27. Периодически (linux — раз в минуту) данные из кэшей сбрасываются на диск — синхронизируются. Можно насильно, «добрым словом и пистолетом» заставить ОС сбросить кэши на носитель командой sync. Кстати, эта команда «скрытно» выполняется, когда мы указываем операционной системе «отсоединить/извлечь флэшку».

Параметры создания файловых систем (процесса форматирования раздела) в linux обычно указываются в конфигурационных файлах.

Пример: Файл /etc/mke2fs.conf устанвливает основные параметры файловой системы ext2/3/4 при форматировании раздела (см. рис. 28).

```
[defaults]
base_features = sparse_super,large_file,filetype,resize_inode,dir_index,ext_attr
default_mntopts = acl,user_xattr
enable_periodic_fsck = 0
blocksize = 4096
inode_size = 256
inode_ratio = 16384

[fs_types]
ext3 = {
        features = has_journal
}
ext4 = {
        features = has_journal,extent,huge_file,flex_bg,uninit_bg,dir_nlink,extra_isize
        auto_64-bit_support = 1
        inode_size = 256
}
ext4dev = {
        features = has_journal,extent,huge_file,flex_bg,uninit_bg,dir_nlink,extra_isize
```

```
		inode_size = 256
		options = test_fs=1
	}
	small = {
		blocksize = 1024
		inode_size = 128
		inode_ratio = 4096
	}
	floppy = {
		blocksize = 1024
		inode_size = 128
		inode_ratio = 8192
	}
	big = {
		inode_ratio = 32768
	}
	huge = {
		inode_ratio = 65536
	}
	news = {
		inode_ratio = 4096
	}
	largefile = {
		inode_ratio = 1048576
		blocksize = -1
	}
	largefile4 = {
		inode_ratio = 4194304
		blocksize = -1
	}
	hurd = {
	     blocksize = 4096
	     inode_size = 128
	}
```

Рис. 28. Файл /etc/mke2fs.conf

3.4. Блок и кластер

Блок файловой системы — небольшая последовательность смежных секторов, обычно размером 1, 2, 4, 8, 16, 32, 64 сектора. Изредка используется ещё размер 128 секторов в блоке. Размер блока определяется при форматировании раздела программой mkfs. Блок - логическая единица хранения данных в unix-овых файловых системах.

Цель появления этого понятия:

- уменьшить числа (адреса секторов) с которыми приходится оперировать алгоритмам файловой системы;
- тем самым, ускорить работу файловой системы;
- увеличить размеры обрабатываемых файловой системой разделов, файлов.

Как было сказано в первой лекции, количество секторов на современных дисках уже достигло десятков миллиардов и продолжает расти. То есть, имеем числа, которые в обычный 32-разрядный калькулятор уже не влазят. То есть, 32-разрядный процессор неспособен их обработать одной командой, как числа с фиксированной точкой. А обрабатывать эти числа арифметикой чисел

с плавающей точкой (как «действительные») - нельзя, ибо в адресной арифметике всегда нужен точный результат. Конечно существуют алгоритмы работы со сверхбольшими числами, но они работают на многие порядки медленнее, чем арифметика чисел с фиксированной точкой. Чтобы решить эту проблему — очень большое количество секторов на новых винтах, было введено понятие блок. То есть, определяем блок в 8 секторов, тем самым уменьшаем адрес в 8 раз, что уже может помочь.

Специфика понятия «блок». Блок в unix`овых файловых системах (UFS, ext и некоторых других) может делиться на фрагменты блока. Фрагмент — последовательность смежных секторов в блоке. Фрагментов в блоке может быть 2, 4, 8 — в зависимости от размера блока. Придуманы фрагменты для того, чтобы уменьшить потери пространства в файловой системе, поскольку последний (или единственный) блок файла обычно неполный. Файловая система может помещать маленькие файлы во фрагменты блока, тем самым экономя пространство.

Таким образом, наименьшее место на диске, которое может быть выделено для хранения файла в unix-овых файловых системах — один фрагмент блока.

Например, в файловой системе ext2/3/4 размер блока в настоящее время (ноябрь 2019 года, версии ядра 4.x, 5.x) определён в файле /*etc*/mke2fs.conf и составляет 1, 2, 4 или 8 кб, то есть, 1, 4, 8 или 16 секторов.

Кластер файловой системы — последовательность смежных секторов, длиной 1, 2, 4, 8, 16, 32, 64 и изредка может быть 128 байт. Логическая единица хранения данных в MS DOS и Windows. Как правило, это наименьшее место на диске, которое может быть выделено для хранения файла в Виндовых файловых системах.

Цель появления этого понятия такая же, как и для понятия блок файловой системы (см. выше). Но в отличие от блока, кластер не делится на фрагменты, поэтому в Виндовых файловых системах потери дискового пространства заведомо больше.

3.5. Типы файлов в unix

ФС может содержать следующие типы файлов:

- каталог,
- блоковое устройство,
- символьное устройство,
- файл fifo — файл имени именованного канала (именованного pipe),
- файл socket — файл имени сокета,
- символьная (мягкая) ссылка,
- обычный файл.

Файлы блокового и символьного устройств (*файлы устройства)* — файлы нулевого размера, в которых резидентно хранятся адреса драйверов устройств. В них нет ничего интересного, они находятся в каталоге /dev, ими пользуется операционная система. Никаких данных в этих файлах нет. Тем не менее, авторы иногда наблюдают попытки студентов найти свои файлы на только что «вткнутой» флэшке в каком-нибудь файле в каталоге /dev.

Файлы устройств обеспечивают доступ к физическому устройству. При создании такого устройства указывается тип устройства (блочное или символьное), *старший номер* — индекс драйвера в таблице драйверов операционной системы (то есть, какой драйвер обслуживает этот тип устройств и его адрес) и *младший номер* — параметр, передаваемый драйверу, поддерживающему несколько устройств, для уточнения о каком «подустройстве» идет речь (например, о каком из нескольких IDE-устройств или COM-портов).

Файлы fifo и socket — аналогично файлы нулевого размера, в которых резидентно хранятся адреса структур в операционной системе, соответственно, именованного канала или сокета, то есть, структур для канального механизма взаимодействия процессов. Создаются

взаимодействующими процессами и ими же используются для передачи друг другу информации. Именованный канал (fifo, именованный pipe), адресуемый файлом имя.fifo, используется для передачи данных между процессами, выполняющимися в одной ОС (локальное взаимодействие), работает по принципу двунаправленной очереди по алгоритму FIFO. Сокет, адресуемый файлом имя.socket, предназначен для взаимодействия между процессами, выполняющимися в одной ОС. Обратите внимание: в одной ОС (!-локальное взаимодействие), то есть, это не тот сокет, через который происходит взаимодействие в Интернет. Хотя механизм взаимодействия очень похожий. При создании Интернет-сокета файла не создаётся.

Файл символьной ссылки - особый тип файла, содержимое которого — не данные, а путь к какому-либо другому файлу (см. рисунок 30). Для пользователя такой файл в большинстве ситуаций неотличим от того, на который он ссылается: операции чтения, записи и пр. над символьной ссылкой работают так, как если бы они производились непосредственно над тем файлом, на который указывает ссылка. Символьные ссылки могут указывать также и на каталог: в этом случае они «работают» как каталоги.

Символьная ссылка имеет ряд преимуществ по сравнению с жёсткой ссылкой (см. ниже пункт 3.6.2): она может использоваться для связи файлов в разных файловых системах (ведь номера индексных узлов уникальны только в рамках одной файловой системы). Кроме того, существование файла-ссылки совершенно независимо от существования того файла, на который он ссылается, поскольку в ссылке хранится только путь к целевому файлу, и нет никакой привязки к индексному узлу. Поэтому возможно удалять файл и символьную ссылку на него независимо: причём в случае удаления целевого файла символьная ссылка продолжит существовать, но останется «битой», то есть, не позволяющей перейти ни к какому файлу.

Каталог - файл, имеющий структуру таблицы, но хранящийся как список (рис. 29).

I - Индекс файла, порядковый номер строки индексной таблицы, в которой содержатся атрибуты файла, 4 байта	L - Общая длина строки каталога — длина всех полей строки в байтах, 2 байта	t — тип файла, битовое поле, 1 байт	l — длина имени файла в байтах, 1 байт	Имя файла, до 255 байт
. . .	. . .		. . .	. . . .
N = {от 2 до 4 млрд}	16		8	file.txt
. . . .	. . .		. . .	. . .

Рис. 29. Каталог.

Всё остальное — **обычные файлы**. Как правило, обычный пользователь работает, конечно же, с обычными файлами.

3.6. Ссылки на файлы

3.6.1. Мягкая ссылка на файл

«Мягкая ссылка» - softlink» - особый тип файла, в котором сохраняется путь к другому файлу (см. рис. 30). Обычно используется тогда, когда неудобно пользоваться полным или относительным (относительно текущего каталога) именем файла, который лежит где-то там,

иногда даже трудно вспомнить, где он там находится и даже как называется. Чтобы с этим не заморачиваться, в своём каталоге создаём мягкую ссылку, в которой это всё запоминается. Создаётся командой:

ln -s <путь_к_файлу_где-то_там> <новое_удобное_имя_здесь>

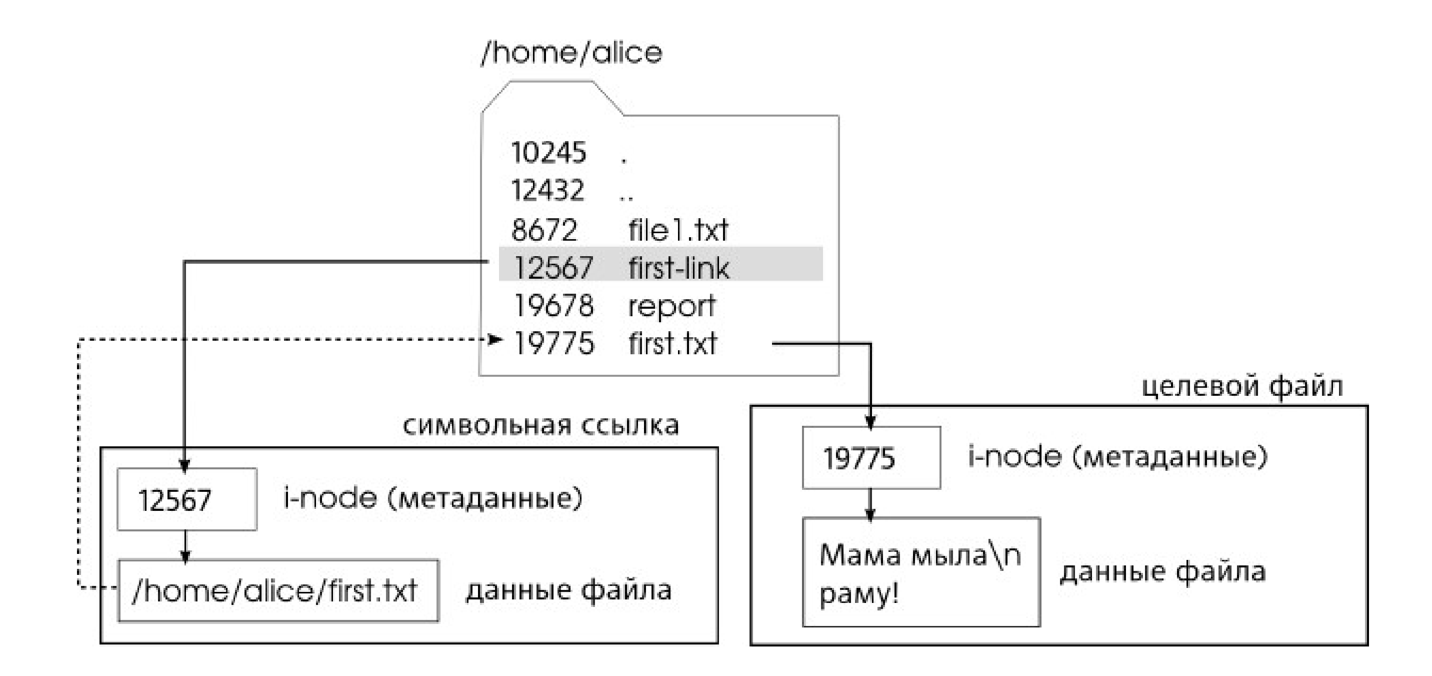


Рис. 30. Пример символьной ссылки на файл: ln -s first-link /home/alice/first.txt

3.6.2. Жёсткая ссылка на файл

В unix-овых файловых системах файл может иметь много имён (очень много имён). Каждое имя файла — жёсткая ссылка (см. рис. 31).

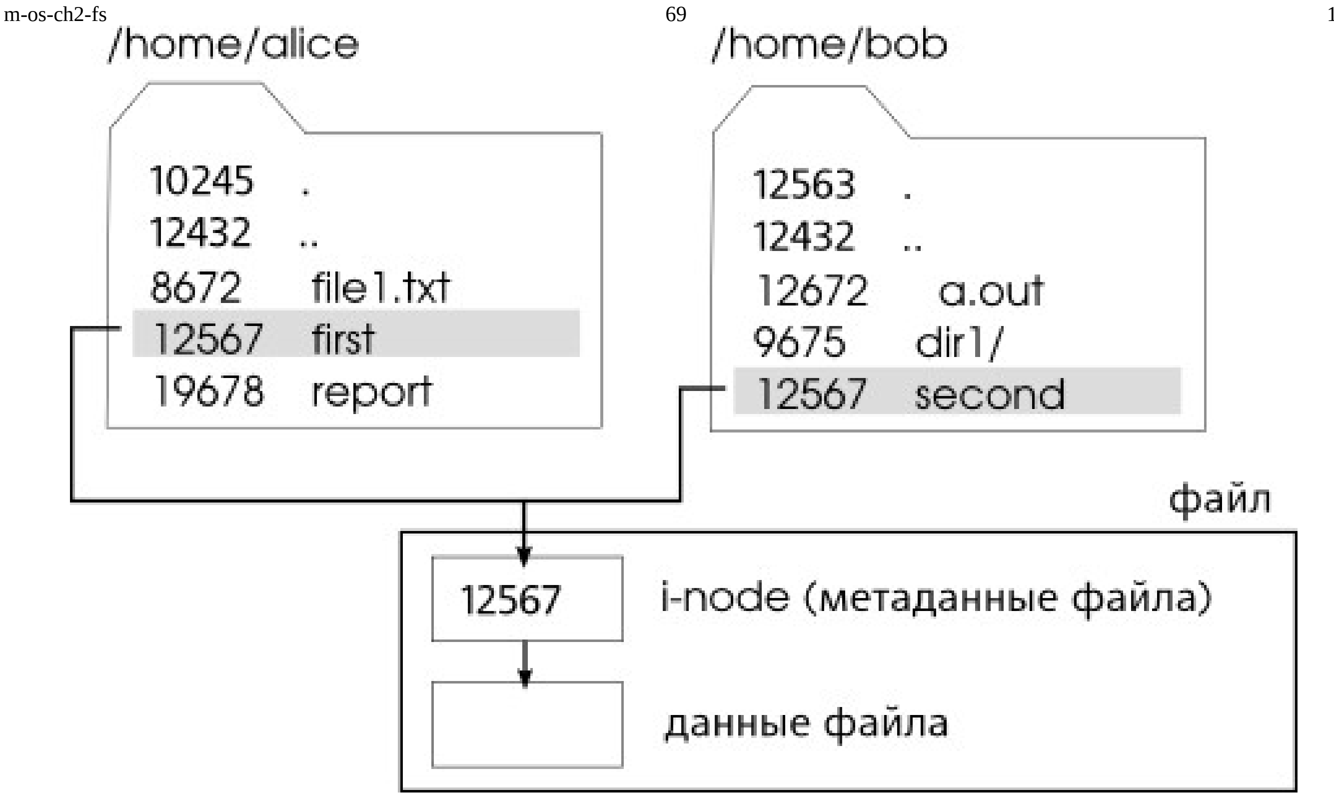


Рис. 31. Жёсткая ссылка — ещё одно имя файла: ln first second

Индекс файла first равен индексу файла second. Индекс файла — это номер строки в индексной таблице, в которой файл описывается, то есть, в этой строке содержатся атрибуты файла, включая поля с адресами блоков в файловой системе, в которых расположены данные файла. Таким образом, строка 12567 индексной таблицы определяет файл, у которого два имени, одно в каталоге /home/alice, а другое в каталоге /home/bob. Совершенно не обязательно, чтобы эти имена были в разных каталогах, можно и в одном каталоге завести несколько имён у некоторого файла. При такой схеме физическое удаление данных файла с носителя происходит только тогда, когда удаляется последняя из существующих жёстких ссылок на этот файл (последнее имя).

В файловых системах unix/linux атрибуты файлов хранятся в строках индексной таблицы, а имена файлов отдельно в каталогах. В строке каталога вместе с именем файла хранится также номер строки индексной таблицы, в которой файл описан. Этот номер называется индексом файла. С помощью этого номера осуществляется связь каталога с индексной таблицей, называется эта связь — link или «жёсткая ссылка».

Такой способ определения файлов (атрибуты отдельно в индексной таблице, имя файла отдельно в каталоге) позволяет одному и тому же файлу присваивать несколько имён. То есть, связей link - «жёстких ссылок», у файла может быть много. Другими словами, у файла может быть много имён. В индексной таблице среди атрибутов файла есть параметр — «количество имён файла», значение этого параметра равно количеству имён файла.

Как это увидеть — см. рис. 32.

```
student@comp10: /home/student
Файл Правка Вид Поиск Терминал Справка
[student@comp10 ~]$ ls -i -l
итого 28
 658661 drwxr-xr-x  3 student student 4096 ноя 19 13:01 docs
 676572 -rw-r--r--  1 student student    0 ноя 26 16:11 file.txt
 656241 drwxr-xr-x  2 student student 4096 окт 28 14:14 public_html
1071142 drwxrwxrwx  2 student student 4096 ноя 12 14:58 tmp
 655174 drwxr-xr-x 18 student student 4096 ноя 11 10:01 ulsu1920
 654131 drwxr-xr-x  6 student student 4096 окт 21 13:02 Документы
 654130 drwxr-xr-x  2 student student 4096 окт 28 15:30 Загрузки
 654129 drwxr-xr-x  5 student student 4096 ноя 19 15:32 Рабочий стол
[student@comp10 ~]$
[student@comp10 ~]$ ln file.txt file1.txt
[student@comp10 ~]$ ln file.txt file2.txt
[student@comp10 ~]$ ln file2.txt file3.txt
[student@comp10 ~]$ ls -i -l
итого 28
 658661 drwxr-xr-x  3 student student 4096 ноя 19 13:01 docs
 676572 -rw-r--r--  4 student student    0 ноя 26 16:11 file1.txt
 676572 -rw-r--r--  4 student student    0 ноя 26 16:11 file2.txt
 676572 -rw-r--r--  4 student student    0 ноя 26 16:11 file3.txt
 676572 -rw-r--r--  4 student student    0 ноя 26 16:11 file.txt
 656241 drwxr-xr-x  2 student student 4096 окт 28 14:14 public_html
1071142 drwxrwxrwx  2 student student 4096 ноя 12 14:58 tmp
 655174 drwxr-xr-x 18 student student 4096 ноя 11 10:01 ulsu1920
 654131 drwxr-xr-x  6 student student 4096 окт 21 13:02 Документы
 654130 drwxr-xr-x  2 student student 4096 окт 28 15:30 Загрузки
 654129 drwxr-xr-x  5 student student 4096 ноя 19 15:32 Рабочий стол
[student@comp10 ~]$
```

Рис. 32. Hardlinks. Стрелка снизу указывает на колонку «количество имён». Обратите внимание, что файловая система не различает, на какое имя вы создали очередную hardlinks — все имена для неё одинаковы. Также обратите внимание на индексы файла в первой колонке.

3.7. Взаимосвязь файлов в файловой системе

Файлы всех перечисленных типов логически объединены в ациклический граф с однонаправленными ребрами, получающийся из дерева в результате сращивания нескольких терминальных узлов дерева или нескольких его нетерминальных узлов таким образом, чтобы полученный граф не содержал циклов. В нетерминальных узлах такого ациклического графа (т.е. в узлах, из которых выходят ребра) могут располагаться только файлы типов " директория " и "связь" (soft). Причем из узла, в котором располагается файл типа "связь", может выходить только ровно одно ребро. В терминальных узлах этого ациклического графа (т.е. в узлах, из которых не выходит ребер) могут располагаться файлы любых типов (см. рис. 33), хотя присутствие в терминальном узле файла типа "связь" обычно говорит о некотором нарушении целостности файловой системы.

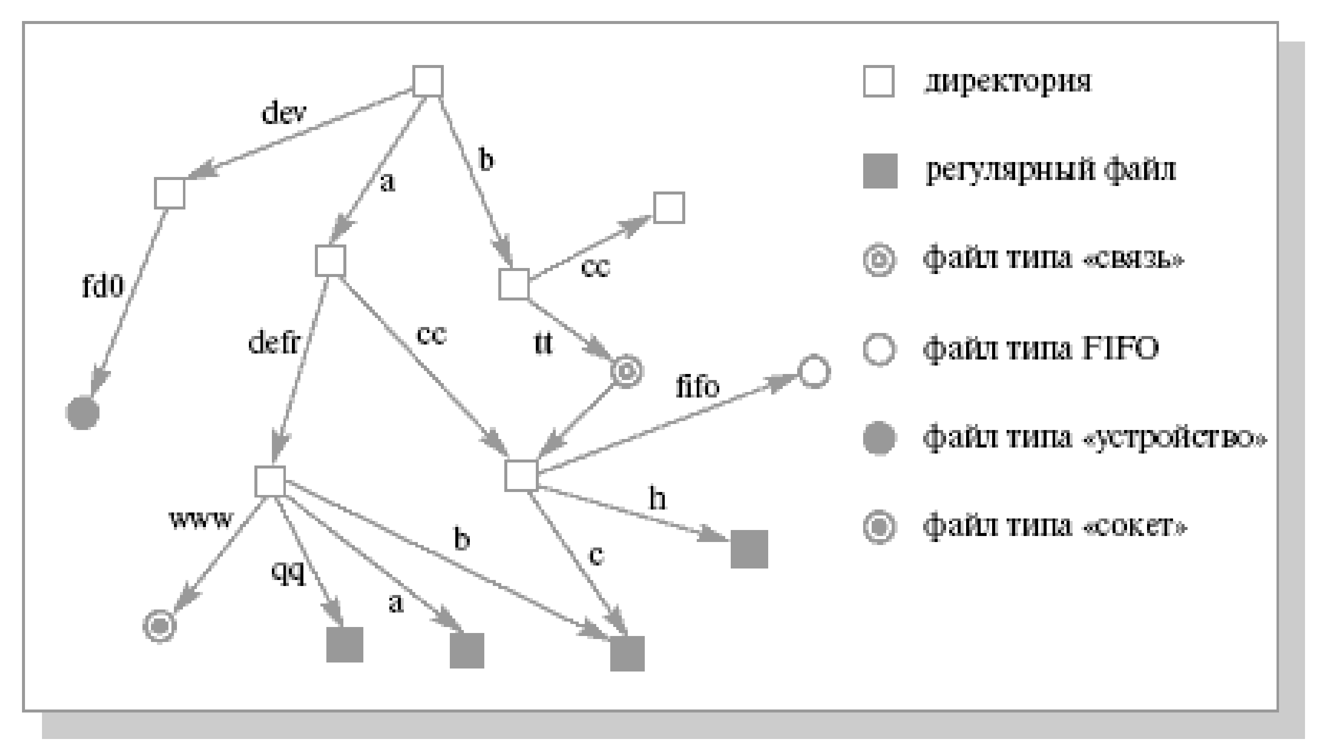


Рис. 33. Пример графа файловой системы.

В отличие от древовидной структуры набора файлов, где имена файлов связывались с узлами дерева, в таком ациклическом графе имя файла связывается не с узлом, соответствующим файлу, а с входящим в него ребром. Ребра, выходящие из узлов, соответствующих файлам типа "связь" (softlink), являются неименованными. Надо отметить, что практически во всех существующих реализациях UNIX-подобных систем в узел графа, соответствующий файлу типа "директория", не может входить более одного именованного ребра, хотя стандарт на операционную систему UNIX и не запрещает этого. В качестве полного имени файла (абсолютного) может использоваться любое имя, получающееся при прохождении по ребрам от корневого узла графа (т.е. узла, в который не входит ни одно ребро) до узла, соответствующего этому файлу, по любому пути с помощью следующего алгоритма:

1. Если интересующему нас файлу соответствует корневой узел, то файл имеет имя " / ".
2. Берем первое именованное ребро в пути и записываем его имя после символа " / ".
3. Для каждого очередного именованного ребра в пути приписываем к уже получившейся строке справа символ " / " и имя соответствующего ребра. То есть, символ " / " - разделитель между именами рёбер (файлов).

Полное имя является уникальным для всей файловой системы и однозначно определяет соответствующий ему файл.

3.8. Восстановление файловой системы

Программа fsck (File System Consistency Check) предназначена для проверки файловых систем и исправления ошибок файловой системы, если они будут обнаружены. Это многопроходная программа контроля файловых систем. Каждый проход файловой системы активизирует различные этапы программы fsck. После инициализации программа fsck выполняет последовательность проходов для каждой файловой системы, проверяя блоки и размеры, полные имена файлов, связность, подсчет ссылок и карту свободных блоков (возможно, перестраивая ее) и выполняет очистку. Может использоваться для проверки всех файловых систем, которые поддерживаются ядром Linux.

Формат вызова программы следующий (из под root):

fsck [параметры] [файловая система]

Последовательность проверки файловой системы должна быть следующая:
1. Размонтировать файловую систему (!).
2. Запустить fsck для ее проверки.
Например, для проверки файловой системы раздела /dev/hda1 сначала размонтируем его, а потом запустим fsck:

```
umount /dev/hda1
fsck /dev/hda1
```

Программа fsck изначально глубоко проверяла все структуры данных подряд, то есть, целиком всю файловую систему. Для поиска ошибок она задействовала методы эвристического анализа для ускорения и оптимизации процесса поиска ошибок. Однако, даже в этом случае для больших по объёму файловых систем эта процедура могла занимать много часов.

Позднее была реализована схема оценки состояния файловой системы, в основе которой лежит признак «чистого бита файловой системы» в суперблоке. Если происходил сбой и файловая система некорректно демонтировалась, то в суперблоке файловой системы не устанавливался этот бит. По-умолчанию в Linux-системах на одном из этапов загрузки системы происходит проверка файловых систем, которые зарегистрированы в файлах /etc/fstab, /etc/vfstab, а также в /etc/filesystems. Таким образом, анализируя «чистый бит» файловой системы во время загрузки системы утилита определяет, стоит ли проводить проверку. В unix/linux в файле /etc/fstab можно указать необходимость проверки файловой системы при загрузке.

Журналируемые файловой системы в настоящее время позволяют утилите работать только с теми структурами данных, которым действительно необходима починка или восстановление. При необходимости fsck может восстановить всю файловую систему целиком благодаря всё тем же журналам файловой системы.

Для Linux-систем довольно часто (в особенности с использованием файловой системы ext) проверка файловой системы может быть организована таким образом, что она будет проводиться при прошествии некоторого числа демонтирований, даже если файловая система полностью исправна. Это особенно актуально для настольных компьютеров, которые могут выключаться/включаться каждые сутки, перезагружаться в связи с особенностью их работы и применения, а также из-за свободного к ним доступа для подключения внешних устройств. В таких случаях проверка файловой системы (хоть и является полезной и благоприятной процедурой), оказывается слишком частой. Последние версии fsck научились делать проверку в фоновом режиме уже после полной загрузки системы.

Вопросы на «засыпку»

1. Можно ли блок данных записать в кластер?

2. Можно ли считать сжатый файл зашифрованным?

3. Является ли архивация файлов сжатием файлов?

4. Если комп начал «тормозить», то чтобы «вылечить» это, достаточно ли подсыпать в кэш процессору витамин B12?

5. Можно ли удалить жёсткую ссылку на файл?

6.

ЛЕКЦИЯ 4. ФАЙЛОВАЯ СИСТЕМА UFS

4.1. UFS — краткое описание: минимум, что надо знать

Unix File System (UFS) — файловая система, созданная для операционных систем семейства BSD и используемая в переработанном и дополненном виде на данный момент как основная в операционных системах-потомках (FreeBSD, OpenBSD, NetBSD, DragonFlyBSD).

Это сложная файловая система, настолько, что, например, в linux её поддержка до сих пор «хромает», а, как известно, поддержка операционной системой linux чужих файловых систем — это показатель. Также следует отметить, что за долгий период эксплуатации алгоритмы этой файловой системы очень хорошо отработаны.

Данная файловая система послужила основой для разработки многих других файловых систем для unix и для unix-подобных ОС, например, linux.

Общая схема файловой системы UFS представлена на рис. 34.



Рис. 34. Раздел с UFS: вторичный загрузчик (загрузочный блок) есть только в первой группе цилиндров.

В UFS используется термин «группа цилиндров» - последовательность смежных цилиндров на hdd. Это обусловлено тем, что UFS возникла давным-давно, в те стародавние времена, когда ЭВМ были большие, а винчестеры маленькие. В смысле, маленькие по объёму дискового пространства, на самом деле, по величине, hdd тоже были большими — размером с письменный стол и весом в сотни килограммов. И в этих маленьких больших hdd использовалась CHS-адресация, отсюда и термин «группа цилиндров». Каждая группа цилиндров — это в сущности, маленький логический диск.

Раздел на hdd это совокупность смежных цилиндров (см. лекцию 2). Смысл разбиения этой «совокупности смежных цилиндров» на ещё более мелкие «совокупности смежных цилиндров» - «группы цилиндров» в том, чтобы минимизировать перемещения головок при доступе к файлам:

при создании файлов их метаинформация (битовые карты, индексная таблица), каталоги, в которых файлы помянуты и сами файлы, как правило, располагаются в одной группе цилиндров — таков алгоритм создания файлов в UFS. Тем самым, повышается вероятность того, что при доступе к некоторому файлу головки придётся перемещать лишь однажды, в крайнем случае, на соседние дорожки. А перемещение и позиционирование головок — это самая медленная операция при доступе и скорость её выполнения за полвека почти не изменилась.

Нулевая группа цилиндров — особая (см. рис. 34): помимо вторичного загрузчика в ней также находится корневой каталог. На рисунке 34 группы цилиндров пронумерованы, начиная с 1, на самом деле счёт идёт, как обычно, с нуля.

В остальных группах цилиндров всё то же самое, только загрузочная область пустая и нет корневого каталога.

Для перестраховки копия суперблока дублируется в каждой группе цилиндров. Загрузочный сектор не дублируется, но по соображениям унификации и единообразия под него просто выделяется место. Таким образом, относительная адресация блоков в каждой группе остается неизменной.

Это означает, что структуры данных файловой системы распределены по разделу. В частности, важнейшая структура файловой системы — суперблок, хранится в стольких экземплярах, сколько определено групп цилиндров, а их, как правило, определяется десятки и сотни на раздел.

Отсюда следует очень высокая надёжность и живучесть данной файловой системы: чтобы её «снести» нужно затереть начальные блоки всех групп цилиндров, а их могут быть сотни на разделе. Если останется хотя бы одна целая группа цилиндров, то можно восстановить файловую систему. Да, при этом можно потерять некоторые файлы, возможно, много файлов, но файловая система будет восстановлена.

В последних версиях этой файловой системы также может поддерживаться журналируемость, что ещё более повышает надёжность и живучесть этой файловой системы

При форматировании раздела вначале определяется размер блока файловой системы. Он устанавливается либо по умолчанию, либо указывается в параметрах команды mkfs из ряда 1, 2, 4, 8, 16, 32, 64 кб. В файловых системах UFS размер блока по умолчанию принимался равным 16 кб, а в UFS2 на новых терабайтных hdd (с 4-килобайтными секторами) был увеличен до 32 кб. Существенным отличием блока UFS является использование фрагментов блока для решения проблемы неэффективного использования свободного пространства. Размер фрагмента обычно равен 2 или 4 кб, но есть ограничение: фрагмент может быть равен 1/8, 1/4, 1/2, либо всему блоку, что эквивалентно отказу от функции фрагментирования блока.

Далее вычисляется количество блоков файловой системы.

Далее раздел делится на «группы цилиндров». Размер группы цилиндров в блоках и, соответственно, количество групп цилиндров в разделе, определяется исходя из размеров блока: битовая карта свободных строк в индексной таблице и битовая карта свободных блоков должны занимать либо полный блок, либо место, кратное блоку, либо определённую его часть (половину, четверть, одну восьмую) - указывается в параметрах команды mkfs. В начале каждой группы цилиндров резервируется место под вторичный загрузчик. Размер этой загрузочной области в UFS был равен 8 кб (независимо от принятого размера блока), а в UFS2 он стал переменным от 0 до 256 кб. Но помещается вторичный загрузчик только в нулевую группу цилиндров, во всех остальных это место резервируется, но пустует.

Далее после загрузочной области идёт суперблок, затем описатель группы цилиндров, затем две битовые карты, затем часть индексной таблицы.

Строка индексной таблицы, обычно называемая inode (одна строка = одна inode, длина 128 байт), она же индексный дескриптор, она же описатель файла, представлена на рисунке 35, а на рисунке 36 — её описание в исходном тексте операционной системы. На рисунке 37 представлено описание каталога в исходниках UFS. Типы полей определены в хидерах.

По умолчанию на каждые 4 блока файловой системы создаётся одна строка индексной таблицы. Это соотношение следует из того, что в среднем один файл в UFS занимает 4 блока. Конечно, за 40 лет средний размер файла вырос. Чтобы это компенсировать, вырос и «умолчательный» размер блока при создании файловой системы и сейчас в UFS2 составляет 32 Кб. Причём, поскольку блок делится на 8, 4 или 2 фрагмента, то при блоке 32 Кб, фрагмент может быть равен 4, 8 или 16 Кб.

При создании файловой системы UFS часть раздела диска резервируется на всякий случай для пользователя root. Эта доля задается параметром free при запуске программы mkfs и по умолчанию составляет от 4 до 10% общего объема файловой системы. Ко "всяким случаям" относят переполнение диска (то есть, превышение выделенного лимита пространства размещенными на разделе файлами), необходимость срочно найти свободный дисковый блок, когда диск почти заполнен, и тому подобное. Например, при входе пользователя в систему программами login, shell и другими, формирующими операционную среду пользователя, создаются временные файлы, а если файловая система переполнена, то возникает «облом» - пользователь не может даже войти в систему. Поэтому, для пользователя root создаётся резерв — ведь, в отличие от обычного пользователя, root должен иметь возможность войти в систему всегда, даже при переполнении диска. В современных системах, где файловые системы размером в сотни гигабайт обычное явление, 4 - 10% свободного объема составляет значительный объем дискового пространства. Поэтому, при создании файловой системы, есть смысл уменьшить этот резерв до «разумного», например, до 1-3 процентов.

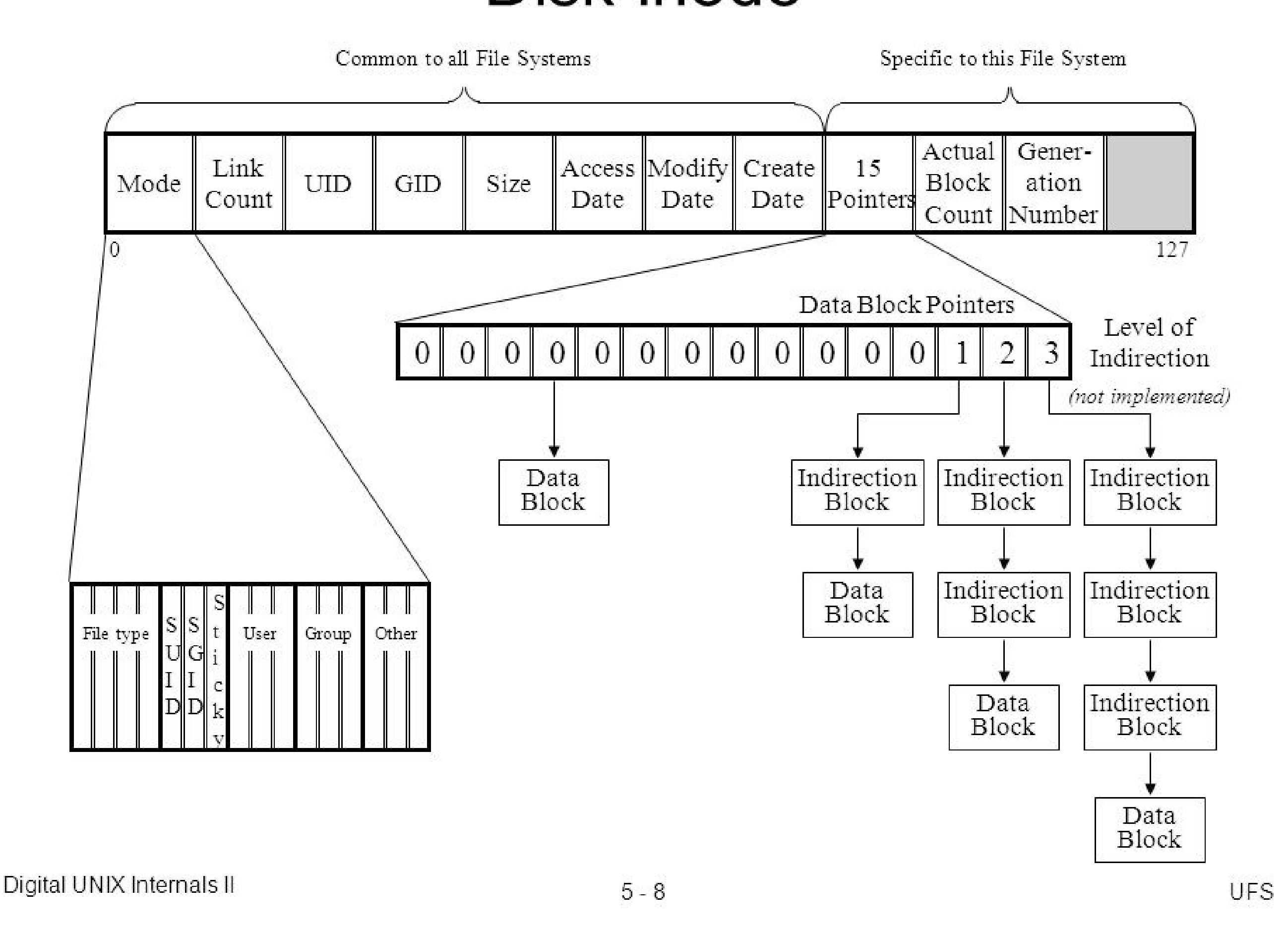


Рис. 35. Содержание строки индексной таблицы.

```
struct ufs1_dinode {
	u_int16_t	di_mode;		/* Разрешения и тип файла */
	int16_t		di_nlink;		/* Количество ссылок */

	union {
		u_int16_t oldids[2];	/* Старые ID пользователя и группы (FFS) */
		} di_u;

	u_int64_t	di_size;		/* Размер файла в байтах */
	int32_t		di_atime;		/* Время последнего доступа */
	int32_t		di_atimensec;		/* Наносекунды */
	int32_t		di_mtime;		/* Время последнего изменения файла */
	int32_t		di_mtimensec;		/* Наносекунды */
	int32_t		di_ctime;		/* Время последнего изменения inode */
	int32_t		di_ctimensec;		/* Наносекунды */
	ufs1_daddr_t	di_db[NDADDR];		/* 12 указателей на прямые блоки */
	ufs1_daddr_t	di_ib[NIADDR];		/* 3 указателя на косвенные блоки */
	u_int32_t	di_flags;		/* Флаги */
	int32_t		di_blocks;		/* Количество выделенных блоков */
	int32_t		di_gen;			/* Поколение */
	u_int32_t	di_uid;			/* ID Владельца */
	u_int32_t	di_gid;			/* ID группы */
	int32_t		di_spare[2];		/* Зарезервировано */
};
```

Рис. 36. Описание строки индексной таблицы в исходниках UFS.

```
struct direct {
	u_int32_t	d_ino;			/* Номер inode */
	u_int16_t	d_reclen;		/* Длина элемента каталога */
	u_int8_t	d_type;			/* Тип файла */
	u_int8_t	d_namlen;		/* Длина имени */
	char		d_name[MAXNAMLEN + 1];	/* Строка с именем файла <= MAXNAMLEN */
};
```

Рис. 37. Описание каталога в исходниках UFS.

Через первое поле каталога (индекс) осуществляется связь между каталогом и индексной таблицей. То есть, индекс файла — это порядковый номер строки индексной таблицы, в которой содержатся атрибуты файла. Длина элемента каталога — это общая длина строки каталога — длина всех полей строки в байтах. По этой длине высчитывается начало следующего элемента каталога. Так как, формально, каталог — это таблица, но хранится как несвязный список (как длинная строка).

Схема адресации блоков файла представлена на рисунке 38. Всего в строке под адресацию выделено 15 полей, из них 12 полей прямой адресации блоков (поле адреса содержит адрес блока) и три блока непрямой адресации.

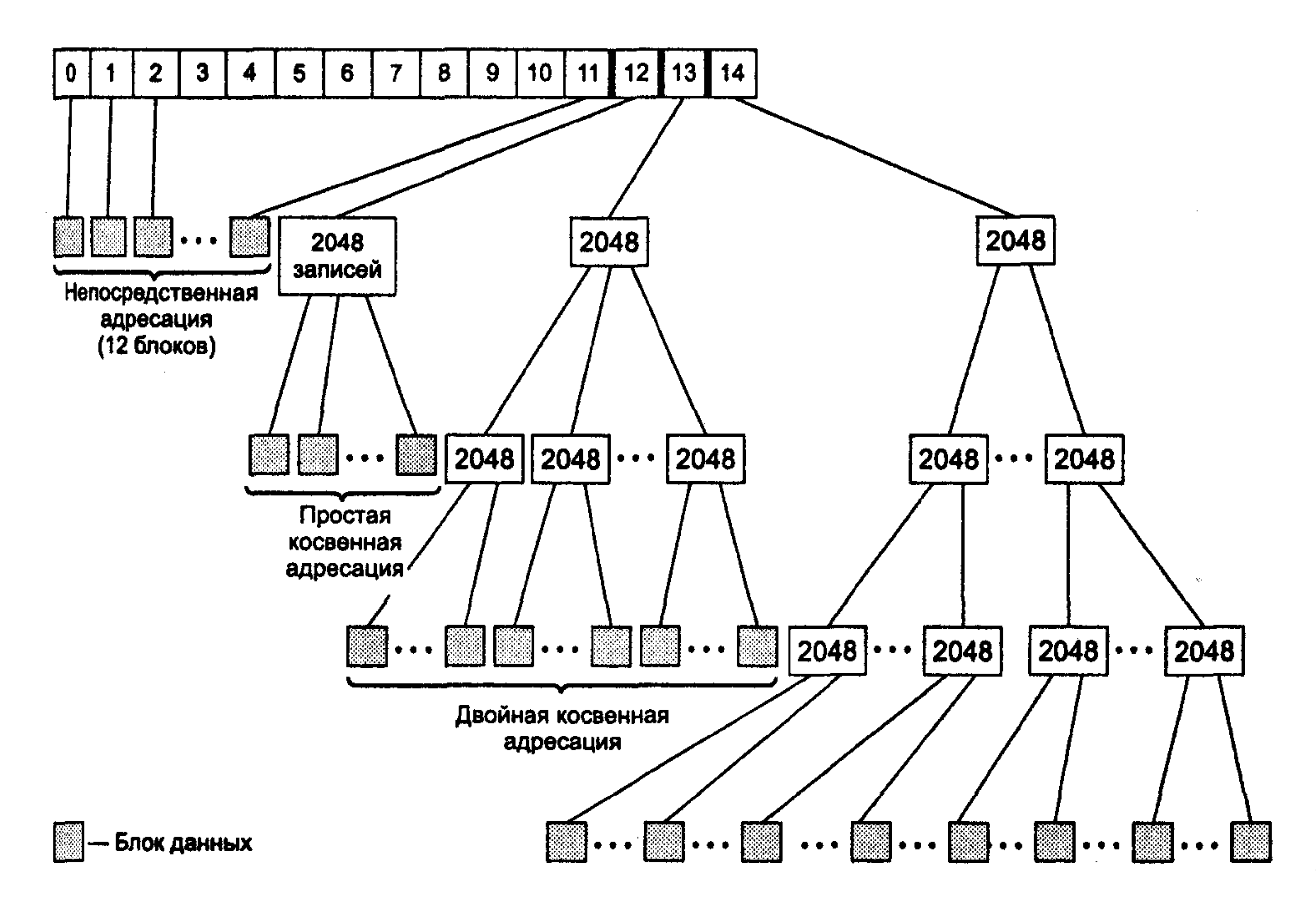


Рис. 38. Схема адресации блоков файла в UFS.

4.2. UFS — более подробное описание

4.2.1. Немного истории

UFS ведет свою историю от S5 FS – самой первой файловой системы, написанной для UNIX в далеком 1974 году. S5 FS была крайне простой и неповоротливой (по некоторым данным, средняя производительность FS составляла всего лишь 2%-5% от «сырой» производительности жесткого диска), но понятия суперблока (super-block), файловых записей (inodes) и блоков данных (blocks) в ней уже существовали (см. рис. 39).

В процессе работы над дистрибутивом 4.2 BSD, вышедшим в 1983 году, S5 FS претерпела некоторые улучшения. Были добавлены длинные имена, символические ссылки и т. д. Так родилась UFS.

В 4.3 BSD, увидевшей свет уже в 1984 году, улучшения носили намного более радикальный, если не сказать революционный, характер. Появились концепции фрагментов (fragments) и групп цилиндров (cylinder groups). То есть, были предприняты меры по консолидации описаний файлов и самих файлов поблизости друг от друга. Благодаря этому, быстродействие файловой системы существенно возросло, что и определило ее название FFS – Fast File System (быстрая файловая система).

Все последующие версии линейки 4.x BSD прошли под знаменем FFS, но в 5.x BSD файловая система вновь изменилась. Для поддержки дисков большого объема и уточнения хранения временных меток ширину полей пришлось удвоить: 32-битная нумерация блоков уступила место 64-битной, а временные метки стали 64-битные. Были внесены и другие менее

существенные усовершенствования.

Поэтому фактически мы имеем дело с тремя различными файловыми системами, не совместимыми друг с другом на уровне базовых структур данных, однако, некоторые источники склонны рассматривать FFS как надстройку над UFS. «UFS (and UFS2) define on-disk data layout. FFS sits on top of UFS (1 or 2) and provides directory structure information, and a variety of disk access optimizations» говорит «Little UFS2 FAQ» (UFS/UFS2 определяет раскладку данных на диске. FFS реализована поверх UFS (1 или 2) и отвечает за структуру директорий и некоторые оптимизации доступа к диску). Если заглянуть в исходные тексты файловой системы, действительно, можно обнаружить два подкаталога – /ufs и /ffs. В /ffs находится определение суперблока (базовой структуры, отвечающей за раскладку данных), а в /ufs – определение inode и структуры директорий, что опровергает данный тезис, с позиций которого все должно быть с точностью до наоборот.

Поэтому, чтобы не увязнуть в болоте терминологических тонкостей, под UFS мы будем понимать основную файловую систему 4.5 BSD, а под UFS2 – основную файловую систему 5.x BSD.

4.2.2. Структура UFS

UFS делит раздел на несколько зон одинакового размера, называемых группами цилиндров (см. также начало пункта 4.1). Каждая зона имеет свою часть индексной таблицы (группу inode) и свою группу блоков данных, независимую от всех остальных зон (см. рис. 39). Другим словами, строки индексной таблицы, размещённые в некоторой зоне (группа inode) описывают блоки данных той и только той зоны, к которой они принадлежат. Это увеличивает быстродействие файловой системы (головка жесткого диска совершает более короткие перемещения) и упрощает процедуру восстановления файлов при значительном разрушении данных, поскольку, как показывает практика, обычно гибнет только первая группа inode — которая находится в начале раздела в нулевой группе цилиндров. Чтобы погибли все группы… ну, это уже сознательные действия.

В UFS каждый блок файловой системы разбит на несколько фрагментов фиксированного размера, предотвращающих потерю свободного пространства в хвостах файлов. Благодаря этому, использование блоков большого размера уже не кажется расточительной идеей, напротив, это увеличивает производительность и уменьшает фрагментацию. Если файл использует более одного фрагмента в двух несмежных блоках, он автоматически перемещается на новое место, в наименее фрагментированный регион свободного пространства. Поэтому фрагментация в UFS очень мала или же совсем отсутствует, что существенно облегчает восстановление удаленных файлов и разрушенных данных.

Файловая система s5

Boot-блок

Суперблок

Список i-узлов

Блоки данных

a)

Файловая система ufs

Boot-блок

Суперблок

Список i-узлов

Блок группы цилиндров

Блоки данных

. . . .

Boot-блок

Суперблок

Список i-узлов

Блок группы цилиндров

Блоки данных

b)

Рис. 39. Структура файловой системы s5 (a) и UFS (b)

Адресация ведется либо по физическим смещениям, измеряемых в байтах и отсчитываемых от начала группы цилиндров (реже – UFS-раздела), либо в номерах фрагментов, отсчитываемых от тех же самых точек. Допустим, размер блока составляет 16 Кб, разбитых на 8 фрагментов. Тогда 69-й сектор будет иметь смещение 512 x 69 == 35328 байт или 1024 x (16/8)/512 x 69 =276 фрагментов.

В начале раздела расположен загрузочный сектор (вторичнй загрузчик), затем следует суперблок, за которым находится одна или несколько групп цилиндров (cg). Для перестраховки копия суперблока дублируется в каждой группе. Загрузочный сектор не дублируется, но по соображениям унификации и единообразия под него просто выделяется место. Таким образом, относительная адресация блоков в каждой группе остается неизменной (см. рис. 40).

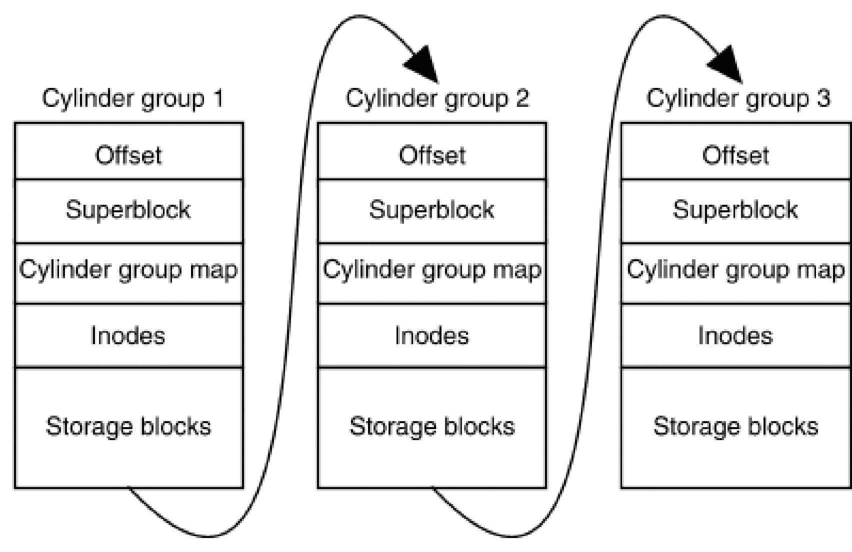


Рис. 40. Последовательно расположенные группы цилиндров

В UFS суперблок располагается по смещению 8192 байт от начала раздела, что соответствует 16-му сектору. В UFS2 он «переехал» на 65536 байт (128 секторов) от начала, освобождая место для дисковой метки и вторичного загрузчика операционной системы, а для действительно больших (в исходных текстах – piggy) систем предусмотрена возможность перемещения суперблока по адресу 262144 байт (целых 512 секторов)!

Среди прочей информации суперблок содержит:

- fs_cblkno – смещение первой группы блока цилиндров, измеряемый в фрагментах, отсчитываемых от начала раздела;
- fs_iblkno – смещение первой inode в первой группе цилиндров (фрагменты от начала раздела);
- fs_dblkno – смещение первого блока данных в первой группе цилиндров (фрагменты от начала раздела);
- fs_ncg – количество групп цилиндров (штуки);
- fs_bsize – размер одного блока в байтах;
- fs_fsize – размер одного фрагмента в байтах;
- fs_frag – количество фрагментов в блоке;
- fs_fpg – размер каждой группы цилиндров, выраженный в блоках (также может быть найден через fs_cgsize).

Для перевода смещений, выраженных в фрагментах, в номера секторов используется следующая формула:

sec_n(fragment_offset) = fragment_offset*(fs_bsize/fs_frag/512)

или ее более короткая разновидность:

sec_n(fragment_offset) = fragment_offset*fs_fsize/512

Структура суперблока определена в файле /src/ufs/ffs/fs.h и в упрощенном виде выглядит так (см. листинг 1).

```
struct fs {
/* 0x00 */ int32_t      fs_firstfield;     /* historic file system linked list, */
/* 0x04 */ int32_t      fs_unused_1;       /* used for incore super blocks */
/* 0x08 */ ufs_daddr_t fs_sblkno;          /* addr of super-block in filesys */
/* 0x0C */ ufs_daddr_t fs_cblkno;          /* offset of cyl-block in filesys */
/* 0x10 */ ufs_daddr_t fs_iblkno;          /* offset of inode-blocks in filesys */
```

```
/* 0x14 */ ufs_daddr_t fs_dblkno;          /* offset of first data after cg */
/* 0x18 */ int32_t      fs_cgoffset;       /* cylinder group offset in cylinder */
/* 0x1C */ int32_t      fs_cgmask;         /* used to calc mod fs_ntrak */
/* 0x20 */ time_t        fs_time;          /* last time written */
/* 0x24 */ int32_t      fs_size;           /* number of blocks in fs */
/* 0x28 */ int32_t      fs_dsize;          /* number of data blocks in fs */
/* 0x2C */ int32_t      fs_ncg;            /* number of cylinder groups */
/* 0x30 */ int32_t      fs_bsize;          /* size of basic blocks in fs */
/* 0x34 */ int32_t      fs_fsize;          /* size of frag blocks in fs */
/* 0x38 */ int32_t      fs_frag;           /* number of frags in a block in fs */
/* these are configuration parameters */
/* 0x3C */ int32_t      fs_minfree;        /* minimum percentage of free blocks */
/* 0x40 */ int32_t      fs_rotdelay;       /* num of ms for optimal next block */
/* 0x44 */ int32_t      fs_rps;            /* disk revolutions per second */
/* sizes determined by number of cylinder groups and their sizes */
/* 0x98 */ ufs_daddr_t fs_csaddr;          /* blk addr of cyl grp summary area */
/* 0x9C */ int32_t      fs_cssize;         /* size of cyl grp summary area */
/* 0xA0 */ int32_t      fs_cgsize;         /* cylinder group size */
/* these fields can be computed from the others */
/* 0xB4 */ int32_t      fs_cpg;            /* cylinders per group */
/* 0xB8 */ int32_t      fs_ipg;            /* inodes per group */
/* 0xBC */ int32_t      fs_fpg;            /* blocks per group * fs_frag */
/* these fields are cleared at mount time */
/* 0xD0 */ int8_t   fs_fmod;               /* super block modified flag */
/* 0xD1 */ int8_t   fs_clean;              /* file system is clean flag */
/* 0xD2 */ int8_t        fs_ronly;         /* mounted read-only flag */
/* 0xD3 */ int8_t   fs_flags;              /* see FS_ flags below */
/* 0xD4 */ u_char fs_fsmnt[MAXMNTLEN];     /* name mounted on */
};
```

Листинг 1. Формат суперблока (второстепенные поля опущены).

За суперблоком находится первая группа цилиндров. В начале каждой группы расположена служебная структура cg (описатель группы цилиндров), содержащая магическую последовательность 55h 02h 09h, позволяющая находить уцелевшие группы цилиндров даже при полном разрушении суперблока (если же суперблок цел, стартовые адреса всех последующих групп вычисляются путем умножения номера группы на ее размер, содержащийся в поле fs_cgsize).

Другие важные параметры описателя группы цилиндров:

- **cg_cgx** – порядковой номер группы, отсчитываемый от нуля;
- **cg_old_niblk** – количество inode в данной группе;
- **cg_ndblk** – количество блоков данных в данной группе;
- **csum** – количество свободных inode и блоков данных в данной группе;
- **cg_iusedoff** – смещение карты занятых inode, отсчитываемое от начала данной группы и измеряемое в байтах;
- **cg_freeoff** – смещение карты свободного пространства (байты от начала группы).

Структура cg определена в файле /src/ufs/ffs/fs.h и выглядит следующим образом (см. листинг 2).

```
#define CG_MAGIC 0x090255
#define MAXFRAG  8
struct cg {
/* 0x00 */ int32_t  cg_firstfield;         /* historic cyl groups linked list */
/* 0x04 */ int32_t  cg_magic;              /* magic number */
/* 0x08 */ int32_t  cg_old_time;           /* time last written */
/* 0x0C */ int32_t  cg_cgx;                /* we are the cgx'th cylinder group */
/* 0x10 */ int16_t  cg_old_ncyl;           /* number of cyl's this cg */
/* 0x12 */ int16_t  cg_old_niblk;          /* number of inode blocks this cg */
/* 0x14 */ int32_t  cg_ndblk;              /* number of data blocks this cg */
/* 0x18 */ struct  csum cg_cs;              /* cylinder summary information */
/* 0x28 */ int32_t  cg_rotor;              /* position of last used block */
/* 0x2C */ int32_t  cg_frotor;              /* position of last used frag */
/* 0x30 */ int32_t  cg_irotor;              /* position of last used inode */
/* 0x34 */ int32_t  cg_frsum[MAXFRAG];     /* counts of available frags */
/* 0x54 */ int32_t  cg_old_btotoff;        /* (int32) block totals per cylinder */
/* 0x58 */ int32_t  cg_old_boff;           /* (u_int16) free block positions */
/* 0x5C */ int32_t  cg_iusedoff;           /* (u_int8) used inode map */
/* 0x60 */ int32_t  cg_freeoff;            /* (u_int8) free block map */
/* 0x64 */ int32_t  cg_nextfreeoff;        /* (u_int8) next available space */
/* 0x68 */ int32_t  cg_clustersumoff;      /* (u_int32) counts of avail clusters */
/* 0x6C */ int32_t  cg_clusteroff;         /* (u_int8) free cluster map */
/* 0x70 */ int32_t  cg_nclusterblks;       /* number of clusters this cg */
/* 0x74 */ int32_t  cg_niblk;              /* number of inode blocks this cg */
/* 0x78 */ int32_t  cg_initediblk;         /* last initialized inode */
/* 0x7C */ int32_t  cg_sparecon32[3];      /* reserved for future use */
/* 0x00 */ ufs_time_t cg_time;              /* time last written */
/* 0x00 */ int64_t  cg_sparecon64[3];      /* reserved for future use */
/* 0x00 */ u_int8_t cg_space[1];           /* space for cylinder group maps */
 /* actually longer */
```

Листинг 2. Структура описателя группы цилиндров/

Между описателем группы цилиндров и индексной таблицей расположена карта занятых inode и карта свободного дискового пространства, представляющие собой обыкновенные битовые поля. При восстановлении удаленных файлов эти битовые карты очень полезны. Отделяя зерна от плевел, они существенно сужают круг поиска, что особенно хорошо заметно на дисках, заполненных более чем наполовину.

За картами следует индексная таблица (массив inode), смещение которого содержится в поле cg_iusedoff (адрес первой группы inode продублирован в суперблоке).

Назначение основных полей inode:

- di_nlink – количество ссылок на файл (0 означает «удален»);
- di_size – размер файла в байтах;

- di_atime/di_atimensec – время последнего доступа к файлу;
- di_mtime/di_mtimensec – время последней модификации;
- di_ctime/di_ctimensec – время последнего изменения inode;
- di_db – адреса первых 12-блоков данных файла, отсчитываемые в фрагментах от начала группы цилиндров;
- di_ib – адрес блоков косвенной адресации (фрагменты от начала группы).

Сама структура inode определена в файле /src/ufs/ufs/dinode.h и для UFS1 выглядит так (см. листинг 3 и рис. 41):

```
struct dinode {
/* 0x00 */ u_int16_t    di_mode;             /*   0: IFMT, permissions; see below. */
/* 0x02 */ int16_t             di_nlink;     /*   2: File link count. */
/* 0x04 */ union {
           u_int16_t oldids[2];              /*   4: Ffs: old user and group ids. */
           int32_t        inumber;           /*   4: Lfs: inode number. */
    } di_u;
/* 0x08 */ u_int64_t    di_size;             /*   8: File byte count. */
/* 0x10 */ int32_t             di_atime;     /*  16: Last access time. */
/* 0x14 */ int32_t             di_atimensec; /*  20: Last access time. */
/* 0x18 */ int32_t             di_mtime;     /*  24: Last modified time. */
/* 0x1C */ int32_t             di_mtimensec; /*  28: Last modified time. */
/* 0x20 */ int32_t             di_ctime;     /*  32: Last inode change time. */
/* 0x24 */ int32_t             di_ctimensec; /*  36: Last inode change time. */
/* 0x28 */ ufs_daddr_t  di_db[NDADDR];       /*  40: Direct disk blocks. */
/* 0x58 */ ufs_daddr_t  di_ib[NIADDR];       /*  88: Indirect disk blocks. */
/* 0x64 */ u_int32_t    di_flags;            /* 100: Status flags (chflags). */
/* 0x68 */ int32_t             di_blocks;    /* 104: Blocks actually held. */
/* 0x6C */ int32_t             di_gen;       /* 108: Generation number. */
/* 0x70 */ u_int32_t    di_uid;              /* 112: File owner. */
/* 0x74 */ u_int32_t    di_gid;              /* 116: File group. */
/* 0x78 */ int32_t             di_spare[2];  /* 120: Reserved; currently unused */
};
```

Листинг 3. Структура строки индексной таблицы (inode) в USF1.

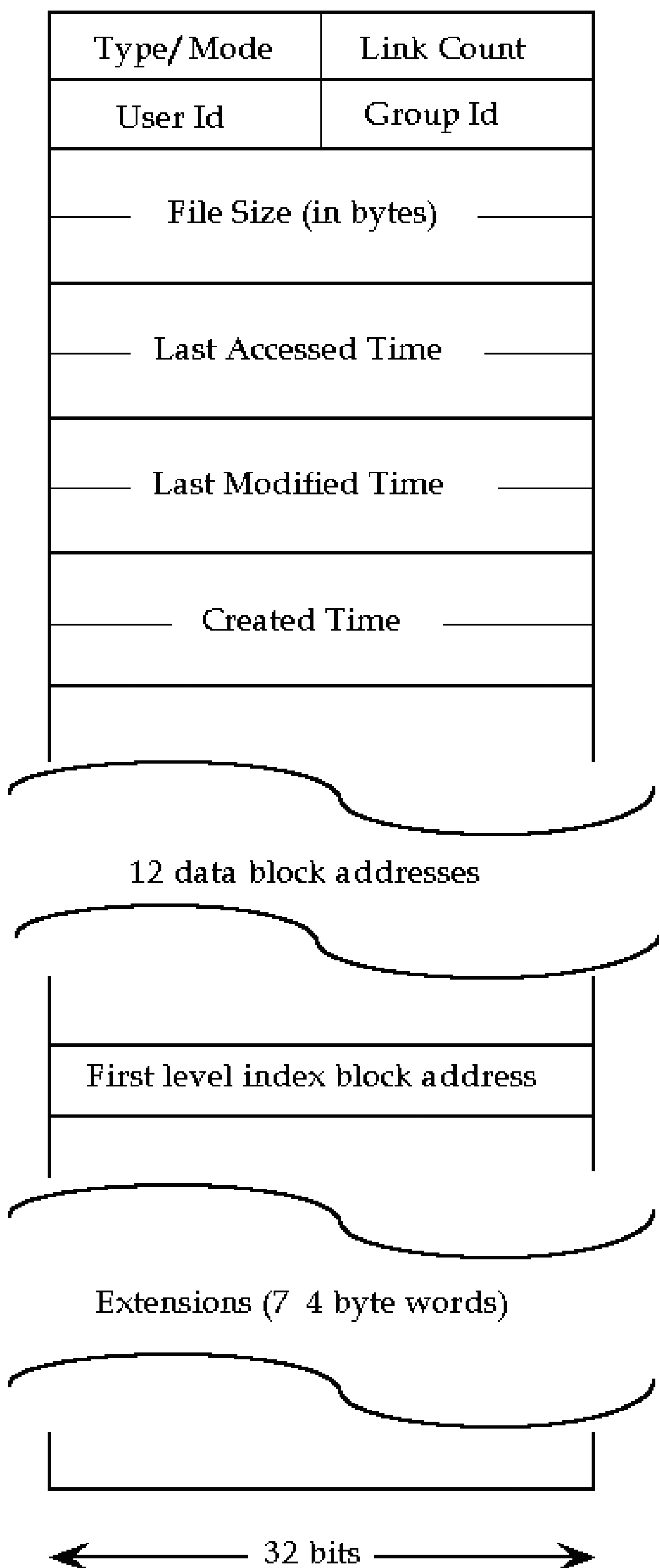


Рис. 41. Схематичное изображение inode. К полю Extensions относится все, что находится ниже di_ib.

В UFS2 формат inode был существенно изменен – появилось множество новых полей, удвоилась ширина адресных полей и т. д. Но в практическом плане изменились только смещения всех полей, а общий принцип работы с inode остался прежним (см. листинг 4).

```
struct ufs2_dinode {
/* 0x00 */ u_int16_t    di_mode;            /*   0: IFMT, permissions; see below. */
/* 0x02 */ int16_t      di_nlink;           /*   2: File link count. */
/* 0x04 */ u_int32_t    di_uid;             /*   4: File owner. */
/* 0x08 */ u_int32_t    di_gid;             /*   8: File group. */
/* 0x0C */ u_int32_t    di_blksize;         /*  12: Inode blocksize. */
/* 0x10 */ u_int64_t    di_size;            /*  16: File byte count. */
/* 0x18 */ u_int64_t    di_blocks;          /*  24: Bytes actually held. */
/* 0x20 */ ufs_time_t   di_atime;           /*  32: Last access time. */
/* 0x28 */ ufs_time_t   di_mtime;           /*  40: Last modified time. */
/* 0x30 */ ufs_time_t   di_ctime;           /*  48: Last inode change time. */
/* 0x38 */ ufs_time_t   di_birthtime;        /*  56: Inode creation time. */
/* 0x40 */ int32_t      di_mtimensec;        /*  64: Last modified time. */
/* 0x44 */ int32_t      di_atimensec;        /*  68: Last access time. */
/* 0x48 */ int32_t      di_ctimensec;       /*  72: Last inode change time. */
/* 0x4C */ int32_t      di_birthnsec;       /*  76: Inode creation time. */
/* 0x50 */ int32_t      di_gen;             /*  80: Generation number. */
/* 0x54 */ u_int32_t    di_kernflags;        /*  84: Kernel flags. */
/* 0x58 */ u_int32_t    di_flags;           /*  88: Status flags (chflags). */
/* 0x5C */ int32_t      di_extsize;         /*  92: External attributes block. */
/* 0x60 */ ufs2_daddr_tdi_extb[NXADDR];     /*  96: External attributes block. */
/* 0x70 */ ufs2_daddr_tdi_db[NDADDR];        /* 112: Direct disk blocks. */
/* 0xD0 */ ufs2_daddr_tdi_ib[NIADDR];        /* 208: Indirect disk blocks. */
/* 0xE8 */ int64_t      di_spare[3];        /* 232: Reserved; currently unused */
 };
```

Листинг 4. Структура inode в USF2.

Имена файлов хранятся в директориях (см. рис. 42). В inode их нет. С точки зрения UFS, директории являются обыкновенными файлами и могут храниться в любом месте, принадлежащем группе цилиндров.

Файловая система UFS поддерживает несколько типов хеширования директорий, однако на структуре хранения имен это никак не отражается. Имена хранятся в блоках, называемых DIRECT BLOCK в структурах типа direct, выровненных по 4-байтовой границе.

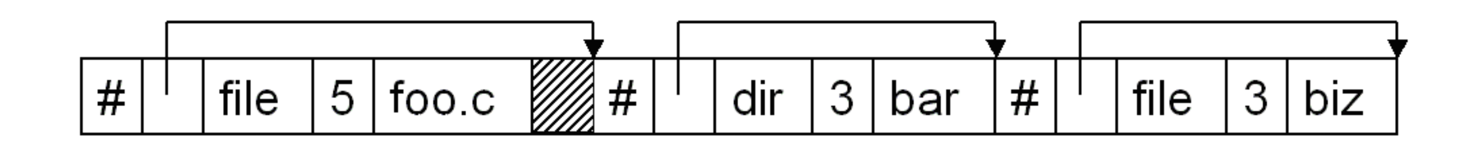


Рис. 42. Хранение имен файлов и директорий.

Структура direct определена в файле /src/ufs/ufs/dir.h и содержит: номер inode (номер строки индексной таблицы), описывающий данный файл, тип файла, его имя, а также длину самой структуры direct, используемую для нахождения следующего direct в блоке.

```
struct  direct {
/* 0x00 */ u_int32_t    d_ino;             /* inode number of entry */
/* 0x04 */ u_int16_t    d_reclen;          /* length of this record */
/* 0x06 */ u_int8_t     d_type;            /* file type, see below */
/* 0x07 */ u_int8_t     d_namlen;          /* length of string in d_name */
/* 0x08 */ char  d_name[MAXNAMLEN + 1];    /* name with length <= MAXNAMLEN */
};
```

Листинг 5. Структура direct, отвечающая за хранение имен файлов и директорий.

4.2.3. Алгоритмы создания и удаления файлов

1. **При создании файла** на UFS-разделе происходит следующее (события перечислены в порядке расположения соответствующих структур в разделе и могут не совпадать с порядком их возникновения):

- в суперблоке обновляется поле fs_time (время последнего доступа к разделу);
- в суперблоке обновляется структура fs_cstotal (количество свободных inode и блоков данных в разделе);
- в группе цилиндров обновляются карты занятых inode и блоков данных – inode, и все блоки данных создаваемого файла помечаются как занятые;
- в inode каталога, в котором создаётся файл, обновляются поля времени последнего доступа и модификации;
- в inode каталога, в котором создаётся файл, обновляется поле времени последнего изменения inode;
- в индексной таблице создаётся (заполняется) строка - inode создаваемого файла: поля di_mode (IFMT – Input Format – формат файла, permissions – права доступа), di_nlink (количество ссылок на файл) и di_size (размер файла);
- в inode создаваемого файла поля di_db (массив указателей на 12 первых блоков файла), и di_ib (указатель на блок косвенной адресации) заполняются в соответствии с размером файла;
- в inode создаваемого файла заполняются поля времени последней модификации и изменения inode, время последнего доступа к файлу (текущим временем);
- в inode создаваемого файла обновляется поле di_spare; в исходных текстах оно помечено как «Reserved; currently unused», но просмотр дампа показывает, что это не так; здесь хранится нечто вроде последовательности обновления (update sequence), используемой для контроля целостности indoe, однако это только предположение;
- в конец директории создаваемого файла дописывается новая структура direct, соответствующая параметрам создаваемого файла.

2. **При удалении файла** на UFS-разделе происходит следующее (события перечислены в порядке расположения соответствующих структур в разделе и могут не совпадать с порядком их возникновения, причём, действия по реальному удалению файла выполняются только если поле di_nlink равно 1):

- в суперблоке обновляется поле fs_time (время последнего доступа к разделу);
- в суперблоке обновляется структура fs_cstotal (количество свободных inode и блоков данных в разделе);
- в группе цилиндров обновляются карты занятых inode и блоков данных – inode, и все блоки данных удаляемого файла помечаются как освобожденные;

- в inode каталога, в котором создаётся файл, обновляются поля времени последнего доступа и модификации;
- в inode каталога, в котором создаётся файл, обновляется поле времени последнего изменения inode;
- в inode удаляемого файла поля di_mode (IFMT – Input Format – формат файла, permissions – права доступа), di_nlink (количество ссылок на файл) и di_size (размер файла) варварски обнуляются (замечание: если счетчик ссылок был больше единицы, то при удалении одной из таких ссылок поле di_nlink уменьшается на единицу, но удаления самого файла не происходит);
- в inode удаляемого файла поля di_db (массив указателей на 12 первых блоков файла), и di_ib (указатель на блок косвенной адресации) безжалостно обнуляется;
- в inode удаляемого файла обновляются поля времени последней модификации и изменения inode, время последнего доступа при этом остается неизменным;
- в inode удаляемого файла обновляется поле di_spare. В исходных текстах оно помечено как «Reserved; currently unused», но просмотр дампа показывает, что это не так. Здесь хранится нечто вроде последовательности обновления (update sequence), используемой для контроля целостности inode, однако это только предположение;
- в директории удаленного файла размер предшествующей структуры direct увеличивается на d_reclen, в результате чего она как бы «поглощает» имя удаляемого файла, однако его перезаписывания не происходит, во всяком случае оно уничтожается не сразу, а только тогда, когда в этом возникнет реальная необходимость.

То есть, как видите, объём работы при создании/удалении файла достаточно большой и операции взаимосвязаны, то есть, представляют собой единую транзакцию. Если что-то пойдёт не так, то происходит откат — весь цикл операций не выполняется.

3. Использование фрагментов для размещения файлов. Если на диск записывается файл размером меньше блока, ему выделяется фрагмент блока, но фактически "резервируется" целый блок, и файл, таким образом, имеет резерв для расширения до размера блока. Файлы размером больше блока записываются в свободные (по возможности, последовательные) блоки файловой системы. Однако размер файла не всегда кратен длине блока, поэтому последний из блоков, в которые записан файл, будет, скорее всего, заполнен не до конца. В этот блок в UFS можно записать фрагмент другого файла. В этом смысле "фрагментом" файла в UFS называется часть файла, меньшая по размеру, чем блок. Фрагменты одного файла не могут храниться в разных блоках. Это означает, что файл не может занимать более одного блока неполностью (см. рис. 43). Индексный дескриптор хранит достаточно информации для того, чтобы идентифицировать не только блок, но и номер фрагмента в блоке, когда речь идет о размещении файла на диске.



Рис. 43. Использование фрагментов для размещения файлов.

4.2.4. Восстановление удалённого файла

Кстати, это возможно, но не просто. После непреднамеренного удаления одного или нескольких файлов необходимо немедленно отмонтировать раздел, чтобы прекратить на него запись, ибо, как описано в пункте 4.2.3., при удалении файла строка индексной таблицы (inode)

обнуляется и в карте свободных inode помечается как свободная. **И, самое главное, блоки, занятые удалённым файлом в битовой карте свободных блоков также помечаются как свободные.** А, значит, при создании нового файла они могут быть использованы для его размещения и тогда данные удалённого файла будут точно затёрты новой информацией.

Далее нужно найти удалённый файл. Поскольку никакой информации о том, где он был расположен в системе не сохраняется, искать файл нужно методом поиска «по образцу». Для этого можно воспользоваться дисковым редактором, работающим на уровне секторов. В этом случае, при наличии достаточного количества свободного места, можно скопировать раздел в файл и открыть его с помощью любого hex-редактора (например, biew) или обратиться непосредственно к самому устройству раздела (например, /dev/ad0s1a).

Поскольку при удалении файла ссылки на 12 первых блоков и 3 блока косвенной адресации необратимо затираются, автоматическое восстановление данных невозможно в принципе. Найти удаленный файл можно только по его содержимому. Искать, естественно, необходимо в свободном пространстве. Вот тут-то и пригодятся битовые карты, расположенные за концом описателя группы цилиндров.

Если повезет и файл окажется нефрагментированным (а на UFS, как уже отмечалось, фрагментация обычно отсутствует или крайне невелика), остальное будет делом техники. Просто выделяем группу секторов и записываем ее на диск, но только ни в коем случае не на сам восстанавливаемый раздел! Например, файл можно передать на соседнюю машину по сети.

К сожалению, поле длины файла безжалостно затирается при его удалении и актуальный размер приходится определять «на глазок». Звучит намного страшнее, чем выглядит. Неиспользуемый хвост последнего фрагмента всегда забивается нулями, что дает хороший ориентир. Проблема в том, что некоторые типы файлов содержат в своем конце некоторое количество нулей, при отсечении которых их работоспособность нарушается, поэтому тут приходится экспериментировать.

А если файл фрагментирован? Первые 13 блоков (именно блоков, а не фрагментов!) придется собирать руками. В идеале это будет один непрерывный регион. Хуже, если первый фрагмент расположен в «чужом» блоке (т.е. блоке, частично занятом другим файлом), а оставшиеся 12 блоков находятся в одном или нескольких регионах. Вообще-то достаточно трудно представить себе ситуацию, в которой первые 13 блоков были бы сильно фрагментированы (поддержка фоновой дефрагментации в UFS , однако, работает). Такое может произойти только при интересной «перегруппировке» большого количеств файлов, что в реальной жизни практически никогда не встречается (ну, разве только, что вы задумали навести порядок на своем жестком диске). Итак, будем считать, что 13-й блок файла найден. В массив непосредственной адресации он уже не помещается (там содержатся только 12 блоков), и ссылка на него, как и на все последующие блоки файла, должна содержаться в блоках косвенной адресации, которые при удалении файла помечаются как свободные, но не перезаписывается, точнее, перезаписывается, но не сразу. Большинство файлов обходятся только одним косвенным блоком, что существенно упрощает задачу.

Как найти этот блок на диске? Вычисляем смещение 13-го блока файла от начала группы цилиндров, переводим его в фрагменты, записываем получившееся число задом наперед (так, чтобы младшие байты располагались по меньшим адресами) и осуществляем контекстный поиск в свободном пространстве.

Отличить блок косвенной адресации от всех остальных типов данных очень легко – он представляет собой массив указателей на блоки, а в конце идут нули. Остается только извлечь эти блоки с диска и записать их в файл, обрезая его по нужной длине.

Внимание! Если вы нашли несколько «кандидатов» в блоки косвенной адресации, это означает, что 13-й блок удаленного файла в разное время принадлежал различным файлам (а так, скорее всего, и будет). Не все косвенные блоки были затерты, вот ссылки и остались. Как отличить «наш» блок от «чужих»? Если хотя бы одна из ссылок указывает на уже занятый блок данных (что легко определить по карте), такой блок можно сразу откинуть. Оставшиеся блоки перебираются вручную до получения работоспособной копии файла. Имя файла (если оно еще не затерто) можно

извлечь из директории. Естественно, при восстановлении нескольких файлов мы не можем однозначно сказать, какое из имен какому файлу принадлежит, тем не менее это все же лучше, чем совсем ничего. Директории восстанавливаются точно так же, как и обыкновенные файлы, хотя в них, кроме имен файлов, нечего и восстанавливать…

Описанный метод восстановления данных страдает множеством ограничений. В частности, при удалении большого количества сильно фрагментированных двоичных файлов он ничем не поможет. Вы только убьете свое время, но навряд ли найдете среди обломков файловой системы что-то полезное. Но как бы там ни было, другого выхода просто нет (если, конечно, не считать резервную копию, которой тоже, скорее всего, нет), поэтому этот метод будет совсем небесполезным.

Конечно, есть «корзина», она теперь перекочевала из MAC`ов и Винды в графические оболочки unix/linux. Но, во-первых, не во все, а во-вторых в unix по-прежнему широко распространён терминальный режим с оболочками командной строки. А в них никаких корзин нет. Поэтому вопрос восстановления удалённых файлов — таки есть.

4.3. Файловая система UFS2 - изменения относительно UFS1

4.3.1. Увеличение размера строки индексной таблицы до 256 байт.

UFS2 стала 64-разрядной, то есть, размер полей увеличился с 4 байт до 8 байт. В результате длина строки индексной таблицы (inode) увеличилась со 128 байт до 256 байт. Однако в целях сохранения преемственности и совместимости разработчики переход на 64-разрядность оформили так, чтобы минимизировать изменения в исходном коде.

Дополнительная информация, необходимая для суперблока и групп цилиндров UFS2, хранится в ранее зарезервированных полях соответствующих структур UFS1. Сохранение одинакового формата этих структур для обеих файловых систем позволяет использовать одну кодовую базу для UFS1 и UFS2. Это избавляет файловую систему от необходимости проверять, с каким именно типом суперблока, группы цилиндров или элемента каталога она работает. Для минимизации количества проверок inodes обоих форматов (индексная таблица) преобразуются при чтении к единому виду, а при записи – обратно. Эффектом этого решения является наличие лишь девяти из нескольких сотен функций, специфичных для UFS1 или UFS2.

Выгода от использования единого кода для обеих файловых систем – значительное уменьшение стоимости сопровождения. Вне девяти чувствительных к формату файловой системы функций разработчики одним движением устраняют ошибки сразу в двух драйверах. Общий код также означает, что поддержка каких либо новых возможностей ОС (таких, как SMP) добавляется только один раз для семейства файловых систем UFS.

Однако, несмотря на то, что в обеих файловых системах все еще используются одинаковые структуры данных для описания групп цилиндров, в UFS2 их определение изменилось. Во времена UFS1 файловая система могла получить точное представление о геометрии диска, включая границы дорожек и цилиндров, точно вычислить круговое положение каждого сектора и, соответственно, вычислить в какой следующий сектор нужно писать информацию, чтобы минимизировать время доступа. Современные диски скрывают эту информацию, предоставляя фиктивные количества блоков на трек, треков на цилиндр и цилиндров на диск, позволяющие, разве что, вычислять размеры разделов. В современных машинах “диск”, предоставляемый файловой системе, в действительности может быть составлен из нескольких физических дисков RAID-массива. Хотя и проводились некоторые исследования по выяснению истинной геометрии диска, сложность эффективного использования такой информации очень высока. Треки на внешней и внутренней кромках современных дисков имеют разное число секторов, и это делает вычисление круговых координат любого конкретного сектора очень сложным. Поэтому в UFS2 было решено избавиться от кода, привязанного к геометрии диска и круговым координатам, и

просто хранить файлы в логически близких друг к другу блоках, что дало даже большую производительность. Старый код обработки групп цилиндров начал удаляться из USF1 в конце 80-х, однако полностью избавиться от него удалось только при разработке UFS2.

4.3.2. Формат каталогов

Файловая система UFS1 использует 32 битные номера inodes. Было соблазнительно увеличить разрядность номера inode до 64 бит, но это привело бы к изменению формата каталога. Но существует очень много кода, работающего непосредственно (а не через интерфейсы) с элементами каталогов.

Изменение формата каталога привело бы к созданию гораздо большего количества специфичных для UFS2 функций, которые увеличили бы сложность сопровождения кода. Кроме того, текущие API используют 32-битные номера inodes для работы с элементами каталогов. Так, даже если бы файловая система поддерживала 64-битные inodes, в настоящее время они были не доступны для приложений. Современные системы даже близко не подходят к ограничению в 4 миллиарда файлов на раздел, налагаемому 32-битным номером inode. Экстраполируя статистику роста количества файлов на разделах за прошедшие годы, можно смело предположить, что 32-битного номера inode будет достаточно еще 10-20 лет. На будущее в суперблоке UFS2 зарезервировано место под флаг, указывающий, что файловая система имеет 64-битные номера inodes.

Также обсуждалась возможность введения более сложной структуры каталога с использованием бинарных структур (B+деревьев) для ускорения операций с большими директориями. Эта технология используется многими современными файловыми системами – XFS, JFS, reiserfs и ext4. Разработчики решили не вносить это усовершенствование по нескольким причинам. Во-первых, они были ограничены во времени и ресурсах, в то время, как стояла задача получить стабильно работающую файловую систему к моменту выпуска FreeBSD 5.0-RELEASE. Сохранив старый формат каталога, программисты смогли использовать весь код работы с директориями из UFS1 и избежали необходимости переписывать различные утилиты файловой системы.

Еще одна причина, по которой было решено сохранить старый формат директорий – динамическое хэширование каталога, которое усовершенствует схему индексирования директории. Что бы избежать повторения линейных поисков в больших каталогах, механизм динамического хэширования при первом обращении к директории на лету строит хэш-таблицу элементов каталога. Эта таблица позволяет избежать сканирования каталога при последующих операциях поиска, создания и удаления. В отличие от файловых систем, изначально спроектированных для работы с большими каталогами, в UFS2 эти структуры не сохраняются на диске, и поэтому файловая система обратно совместима. Благодаря технике динамического хэширования разработчикам удалось существенно снизить влияние больших каталогов на производительность UFS2.

По образцу ext3 был введен флаг, показывающий, поддерживаются или нет дисковые структуры индексирования для каталогов. Этот флаг безоговорочно сброшен в текущей реализации UFS2. В будущем, когда поддержка дисковых структур индексации каталогов будет реализована, драйвер файловой системы не будет сбрасывать этот флаг. Новая файловая система прочитает и использует индексы, оставив флаг установленным. Старая же система, смонтировав однажды этот раздел, сбросит флаг; поэтому, когда управление разделом вновь будет передано новому драйверу, он обнаружит, что индексы не использовались по крайней мере некоторое время и должны быть пересозданы заново. Единственным ограничением в реализации индексов будет то, что они останутся лишь дополнительной структурой, ссылающейся на данные старого линейного формата.

4.3.3. Расширенные атрибуты (extended attributes)

Основное дополнение в UFS2 (относительно UFS1) – это поддержка расширенных атрибутов. Extended attributes (EA) – это часть дополнительного хранилища данных, связанного с inode и используемого для хранения вспомогательных данных, отделенных от содержимого файла. Идея EA концептуально подобна потокам данных, используемым в HFS и HFS+ – файловых системах Apple MacOS и в ntfs в Windows. Но только концептуально. Интегрирование EA в inode позволяет гарантировать их целостность на уровне данных файла. Т.е. успешное завершение вызова fsync() означает, что данные файла, расширенные атрибуты и все имена и пути к файлу корректны и находятся на диске.

Расширенные атрибуты представляют собой пары «имя:значение», которые постоянно связаны с файлами и каталогами, подобно тому как строки окружения связаны с процессом. Атрибут может быть определён или не определён. Если он определён, то его значение может быть или пустым, или не пустым.

Имена атрибутов представляют собой строки с null в конце. Имя атрибута всегда указывается в полной форме — *пространство_имён.атрибут*, например, *user.mime_type*, *trusted.md5sum*, *system.posix_acl_access* или *security.selinux*.

Механизм пространства имён используется для определения различных классов расширенных атрибутов. Эти различные классы существуют по нескольким причинам; например, права доступа и мандаты, требуемые для работы с расширенными атрибутами в одном пространстве имён, могут отличаться в другом.

В настоящее время существует два пространства имен: пользовательское и системное. Пространство имен пользователя не имеет ограничений в отношении именования или содержимого. Системное пространство имен в основном используется ядром для списков контроля доступа и принудительного контроля доступа.

Ядро и файловая система могут ограничивать максимальное количество и размер расширенных атрибутов, которые можно связать с файлом и это (ограничение количества и размера) является очень существенным отличием реализации расширенных атрибутов в файловой системе UFS2 от реализации их в файловой системе ntfs.

В inode UFS1 есть место для хранения двух блоков EA. Новый формат inode в UFS2 резервирует пространство для пяти дополнительных 64-битных указателей. Таким образом, количество блоков EA может быть от 1 до 5. В настоящее время выделено 2 блока для расширенных атрибутов, а 3 других оставлены как запас на будущее (из-за потенциальных проблем с безопасностью). Несмотря на это весь код подготовлен к обработке массива указателей, поэтому, когда количество расширенных атрибутов в будущем переполнит доступное пространство, существующий код будет работать без изменений. Сохраняя 3 блока в запасе, разработчики обеспечили достаточно место для будущего использования. Если пространства под EA понадобится совсем уж много, один из запасных блоков может быть использован для хранения косвенных ссылок на блоки, содержащие указатели на EA.

Первым из двух изначальных применений для EA была поддержка списков контроля доступа (Access Control List – ACL). ACLs заменяют традиционную для UNIX групповую схему доступа к файлу на более специфичную – с двумя списками: пользователей, имеющих доступ к файлу, и конкретных прав для каждого пользователя. Эти права включают как традиционные чтение, запись и исполнение, так и расширенные – право на переименование, удаление файла, и пр.

В ранних версиях ACLs были реализованы в виде единственного на всю ФС дополнительного файла, индексированного по номерам inodes и разбитого на части небольшого размера для хранения ACL-разрешений. Размер каждого такого отрезка был очень мал, так как ACL-файл резервировал место для каждого номера возможного inode. Эта реализация имела 2 недостатка. Фиксированный размер пространства, выделяемого каждому inode, означал, что длинна списка пользователей будет ограничена. Также было сложно атомарно обновлять изменения, вносимые в ACL файла, так как inode файла и блок ACL никак не могли быть записаны одновременно.

Обе проблемы этой реализации были решены хранением ACL-данных в EA-области inode. из-за большого размера EA-области (минимум 8 kb, типично 32 kb) стало легче хранить длинные ACL-списки. Атомарное обновление так же упростилось, так как запись на диск inode приводила к

фиксации всех адресуемых им наборов данных (в том числе и EA областей) в одной дисковой операции, в то время как старый ACL-файл обновлялся только при вызове fsync().

Обратите внимание: поддержка расширенных атрибутов введена так, что их использование сильно ограничено. Расширенный атрибут — это совсем не поток файла в понимании ntfs. В UFS2 расширенный атрибут существенно ограничен и по размеру, и по структуре, и по содержанию в отличие от ntfs, где в поток файла может быть занесена любая информация неконтролируемого размера.

4.3.4. Новые возможности файловой системы

С введением нового формата inode появилась возможность внести несколько дополнительных усовершенствований. Разработчики решили заранее позаботиться о "проблеме 2038" (именно в 2038-м году переполнятся 32-битные поля, хранящие количество секунд, прошедших с начала 1970 года). Поля хранения времен модификации, доступа и изменения были расширены до 64 бит. Это позволяет хранить время с точностью до наносекунды, что особенно актуально для современных производительных и многоядерных процессоров, и поднять границу хранимого времени до 25 апреля 2514 года, а это будет уже 26 век. Доживёт человечество? Вот, хоть бы одним глазком . . ., только напосмотреть.

В тоже время было добавлено новое поля для хранения времени создания файла. Это поле устанавливается при первом размещении inode и не меняется вплоть до его удаления. Оно было добавлено в структуру, возвращаемую вызовом stat() для тех приложений, которым может оказаться полезным это значение (к примеру, различные архивирующие программы, такие, как dump, tar и pax). Время создания было добавлено в ранее зарезервированное поле структуры stat, так что размер структуры не изменился. Таким образом, старые программы смогут также корректно обрабатывать результат вызова stat().

В настояще время только программа dump изменена для использования времени создания файла. Эта новая версия dump, которая может обрабатывать разделы как UFS1, так и UFS2, создает дамп нового формата, не читаемого старой версией restore.

Вызов utimes(), что устанавливает произвольные временные штампы на файл, используется, как правило, при разархивировании файлов для выставления корректных значений времен доступа. Благодаря использованию этих программ может оказаться, что время создания файла будет больше, чем время его модификации. Семантика этого вызова была изменена таким образом, что, хотя он и не позволяет вручную устанавливать время создания файла, но следит за тем, что бы оно никогда не принимало значения меньше, чем меньшее из времен модификации.

Еще одно изменение коснулось поля флагов inode, которое теперь разбито на 2 отдельных 32-битных поля. В первом поле содержатся доступные пользователю флаги, во втором – флаги ядра, такие, как SNAPSHOT или OPAQUE. Таким образом, флаги ядра больше не могут быть изменены вредоносным приложением.

4.3.5. Динамические inodes

Один из основных недостатков UFS1 состоит в том, что она размещает всю индексную таблицу во время создания файловой системы, во время форматирования. Для файловых систем с миллионами файлов процесс инициализации может занимать несколько часов. Кроме того, программа создания новой файловой системы, newfs, предполагает, что каждая файловая система будет заполнена большим количеством мелких файлов и размещает гораздо больше inodes, чем обычно требуется. Если UFS1 использовала все свои inodes, единственный способ получить больше – создать дамп файловой системы , переформатировать раздел с новыми параметрами и восстановить данные из дампа. UFS2 решает эту проблему динамическим размещением inodes. Обычно реализация динамического выделения inodes требует отдельных структур метаданных, отслеживающих текущие наборы inodes. Управление и поддержка этих структур добавляет сложности и отрицательно сказывается на производительности файловой системы.

Для ослабления влияния этих факторов UFS2 условно (нежёстко) резервирует место под индексную таблицу и размещает некоторое количество номеров inodes в блоках каждой группы цилиндров - «ленивое» создание индексной таблицы. Изначально каждая группа цилиндров имеет один блок заранее выделенных inodes (32 или 64 inodes). Когда это блок переполняется, размещается и инициализируется следующий блок inodes. Блоки, которые могут быть выделены под inodes, не являются частью пространства прочих свободных блоков и отдаются под данные файлов только в том случае, если все другие блоки в файловой системе исчерпаны.

Теоретически файловая система может заполнится, использовав все свободные inode-блоки. Позднее, когда будут удалены некоторые большие файлы и на их месте понадобится создать множество мелких, файловая система может обнаружить, что не в состоянии разместить требуемое количество inodes, т.к. все inode-блоки заняты. Может понадобится переразместить существующие файлы (которые оказались в блоках, условно зарезервированных для индексной таблицы) для перемещения их данных в обычные свободные блоки и освобождения места под inodes. Этот код еще не написан, так как подобная ситуация вряд ли возникнет на практике, потому что резерв свободного места на подавляющем большинстве файловых систем (4-8%, напомним, что UFS жестко устанавливает этот предел, и если файловая система заполнена данными ровно на 92%, вызовы, требующие размещения блоков, будут работать только от root, возвращая процессам других пользователей ENOSPC - «нет места на диске») больше требуемого для inodes пространства (26%). На таких системах только процессы, запущенные от root, имеют право размещать inode-блоки. Как только в этом коде появится практическая необходимость, он сразу же будет написан. Пока же файловая система, при возникновении описанных условий, сообщает об исчерпании inodes при попытке создать файл.

Одно из выгодных отличий динамического размещения inodes состоит в том, что время создания UFS2 составляет около 1% от показателей UFS1. Недостаток – необходимость дисковой записи большого куска данных при создании каждого 64-го inode.

4.3.6. Загрузочная область

UFS1 резервирует 8 Кб в начале раздела под загрузочную область раздела — вторичный загрузчик. Хотя это пространство и выглядит огромным в сравнении с 1 Кб, который выделяли файловые системы предыдущего поколения, с другой стороны современные загрузчики требуют все больше и больше места. Поэтому разработчики решили пересмотреть размер загрузочного кода в UFS2. Теперь boot-сектор раздела может иметь размер 0, 8, 64 (по умолчанию) и 256 Кб.

4.3.7. Изменения и усовершенствования в soft updates

Традиционно целостность файловой системы обеспечивается или применением синхронной записи для упорядочивания зависимых обновления данных, либо использованием упреждающего журналирования для атомарной группировки этих обновлений. Soft updates ("мягкое журналирование") - альтернативный подход. Это реализация механизма, который отслеживает зависимости изменений метаданных и организует их фиксацию на диске таким образом, что файловая система всегда остается в непротиворечивом состоянии. Использование мягкого журналирования избавляет от необходимости содержать выделенный дисковый журнал или прибегать к синхронной записи – оба эти подхода отрицательно влияют на производительность.

Добавление к inode расширенных атрибутов заставило внести изменение в код soft updates с тем, что бы обеспечить целостность и этих структур данных. Как и с регулярными блоками (блоками, содержащими данные регулярных файлов), необходимо быть уверенным, что блоки данных и битовые карты, имеющие отношение к EA, зафиксированы до того, как связанные изменения попадут в дисковый inode. Soft updates так же обеспечивают фиксацию блоков данных EA при вызове fsync() на файле.

В связи с этим в текущей реализации soft updates были сделаны два важных усовершенствования. Задуманные изначально для UFS2, они автоматически попали и в UFS1, т.к. код этой части обе ФС разделяют.

Для пользователя удаление файла происходит очень быстро, однако в действительности процесс освобождения inode файла и возвращения всех его блоков в свободный список может занимать несколько минут. Пространство, занимаемое файлом в UFS2, не попадает в статистику файловой системы до тех пор, пока все его блоки не окажутся физически освобожденными. Таким образом, пользователь может удалить несколько файлов, однако освобождение места на диске увидит далеко не сразу. Это может стать серьезной проблемой для программ типа кэша браузера: обнаружив нехватку дискового пространства, он начинает небольшими кусками удалять наиболее старые данные, периодически проверяя сколько освободилось места. И пока данные о действительно свободном пространстве дойдут до программы df, из кэша могут быть удалены и самые последние данные. Для решения подобных проблем механизм soft updates теперь поддерживает специальный счетчик, содержащий количество блоков, удерживаемых файлом в процессе удаления. При вызове statfs() это счетчик добавляется к количеству свободных блоков файловой системы. В результате свободное место на диске появляется сразу после вызова unlink() или завершения работы rm.

Второе изменение касается ложных сообщений о нехватке дискового пространства. Почти заполненная файловая система может отвечать на конкретные запросы о выделении блоков сообщениями о нехватке места, даже если команда df говорит об обратном. Это происходит потому, что soft updates не управляют дисковым пространством, которое удерживают файлы, находящиеся на стадии удаления.

Изначально эту проблему пытались решить, просто позволяя процессам ждать появления свободного места. Однако ждать зачастую приходится около минуты. Кроме того, что такое ожидание делает приложение невыносимо медленным, еще удерживается блокировка на inode, что сказывается также на работе других процессов. Хотя эти условия и не приводят к тупиковой ситуации и снимаются в течение 1-2 минут, пользователи часто думают, что система зависла и жмут на reset.

Для излечения этой болезни в UFS2 было решено кооперировать процессы,ожидающие освобождения блоков и заставлять их работать в помощь soft updates. Планировщик задач отдает кванты времени ожидающих процессов подсистеме soft updates, которая использует их для ускорения фиксации изменений. Таким образом, чем больше процессов в системе ожидают выделения блока, тем быстрее soft updates завершит свою работу, Обычно дело решается в 1-2 секунды.

4.3.8. Проверка целостности большой файловой системы

Обычно после аварийного выключения системы программа fsck проходит по всем inodes и битовым картам, проверяя, какие из inodes и блоков реально используются, и в случае необходимости вносит исправления. Текущая реализация soft udates гарантирует непротиворечивость всех метаданных файловой системы, однако остаются 2 возможные несогласованности: отсутствие в битовой карте пометки об освобождении реально нигде неиспользуемого блока и завышенное количество ссылок у inode. Конечно, использовать файловую систему после сбоя полностью безопасно даже без запуска fsck, однако может потеряться некоторое количество дискового пространства. Поэтому была разработана новая версия fsck, способная работать в фоновом режиме на активной файловой системе для поиска и восстановления потерянных блоков и inodes.

В сочетании со снапшотами эта задача по "сбору мусора" не выглядит такой уж сложной. При работе в фоновом режиме fsck делает снимок файловой системы и в дальнейшем проверяет его как обычную файловую систему. Отыскав все потерянные ресурсы, fsck, через специальный системный вызов, извещает файловую систему о необходимых изменениях и удаляет снапшот.

Кроме того, фоновая проверка большой файловой системы требует много памяти, объем которой растет пропорционально размеру файловой системы. На каждый inode регулярного файла

требуется 4 байта, на inode каталога – 40-50 байт, на блок данных – 1 бит. На типичном разделе UFS2 с 16 Кб блоком и 2 Кб фрагментом fsck нуждается как минимум в 64 Мб памяти на 1 Тб файловой системы. Память, необходимая для проверки inodes, благодаря динамическому размещению, зависит от их количества. В худшем случае, как замечают разработчики, fsck при проверке большого раздела может потребовать гораздо больше памяти, чем реально установлено на машине, а теоретически, может превзойти предельный объем, доступный при 32-битной адресации. Надежда здесь только на то, что с ростом объемов дисков они будут заполняться файлами музыки и видео все больших размеров, что несколько ограничит рост потребления памяти во время работы fsck.

Программисты FreeBSD понимают, что журналируемые файловые системы обеспечивают гораздо более быстрое и экономичное восстановление файловой системы. По этой причине начата работа по добавлению класса GJOURNAL к GEOM и постепенному отказу от использования soft updates. Однако даже журналируемые файловые системы нуждаются в fsck на случай сбоев оборудования или ПО.

По желанию при установке системы в UFS2 может быть включено журналирование.

4.3.9. Производительность

Производительность файловой системы UFS2 приблизительно аналогична UFS1. Это обусловлено прежде всего значительной общностью кодовой базы обеих файловых систем, и тем, что они используют одинаковые алгоритмы. Однако не стоит забывать, что целью создания UFS2 было не кардинальное увеличение производительности, которая уже составляет 80-95% от пропускной способности дисковой подсистемы, а обеспечение поддержки многотерабайтных файловых систем и новые возможностей (таких как EA) без потери производительности.

4.3.10. Планы на будущее

Политика выделения блоков UFS стремиться размещать блоки файлов на диске как можно более непрерывно. Метаданные, описывающие большой файл, в результате состоят из косвенных блоков с множеством последовательных номеров блоков, которые попусту забивают память, так как для открытого файла UFS старается хранить все метаданные в RAM. В UFS2 объем требуемой памяти удвоился, так как указатели на блоки стали 64-битными. Для экономии системной памяти и дискового пространства современные файловые системы используют экстенты – структуры данных, состоящие из пары чисел (адреса начала экстента и его длины), полностью описывающих непрерывный участок выделенного дискового пространства (см. рисунки 43, 44). В случае, когда файл размещен почти непрерывно, достигается существенная экономия системных ресурсов, однако если файл сильно фрагментирован, такой подход создаст еще больше проблем, чем традиционные косвенные блоки. Также стоит обратить внимание, что стоимость произвольного доступа к данным файла при использовании экстента существенно возрастает – может возникнуть необходимость перебрать множество экстентов для достижения нужного логического смещения в файле. В современных файловых системах (таких, как XFS, JFS, reiser4, ext4) эта проблема решается только с помощью индексирования экстентов по логическому смещению с помощью B+деревьев.

Разработчики FreeBSD только планируют внедрение схемы учета блоков файла на экстентах. А пока в inode добавлено поле, которое позволяет файловой системе использовать для конкретного файла больший размер блока. Для маленьких, слабо растущих или сильно фрагментированных файлов в этом поле устанавливается единица. А когда файловая система обнаруживает большой непрерывный файл, она пишет сюда число от 2 до 16, на которое домножается размер блока файловой системы при работе с метаданными этого файла. Недостаток подхода очевиден – исчерпав блоки большого размера, файловая система будет возвращать ENOSPC при попытке увеличить такой файл. Возможно, будет написан код, пересчитывающий все метаданные такого файла для уменьшенного размера блока. Эта процедура будет вызывать

длинную паузу, однако предполагается, что подобные экстремальные условия будут возникать крайне редко.

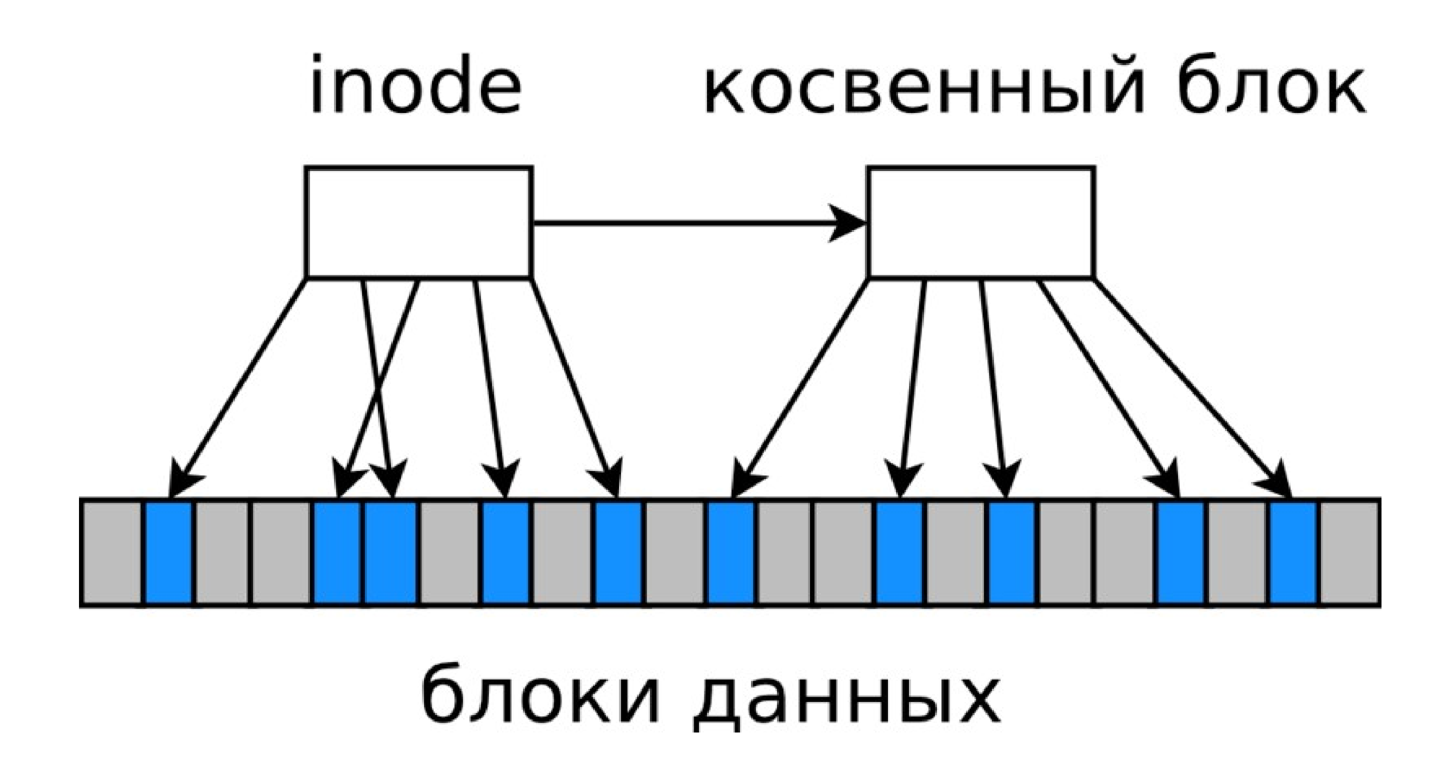


Рис. 44. Файл, размещённый без использования экстентов, «по старинному».

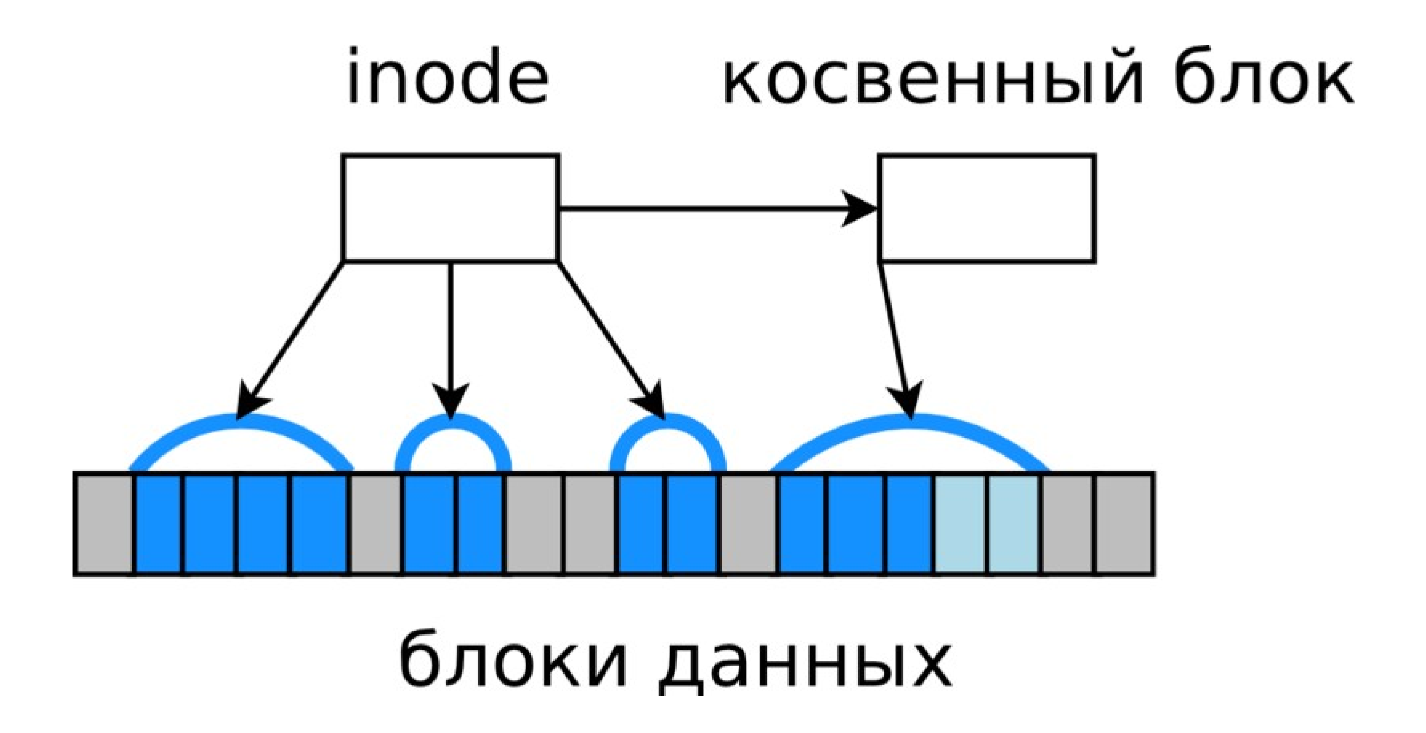


Рис. 45. Файл, размещённый с использованием экстентов. В конце голубым помечены блоки ещё не занятые данными файла, но зарезервированные для него.

Вопросы на «засыпку»

1. В файловой системе UFS блоки делятся на фрагменты. Как это влияет на фрагментацию файлов?

2. Предположим, что раздел размером 100 Gb отформатирован с параметрами по умолчанию с размером блока 16 Кб, фрагмент 4 Кб (то есть, в расчёте на среднестатистическое применение). Но так получилось, что нужно сохранить на этом разделе только мелкие файлы средним размером менее 4-х килобайт в количестве 10 млн штук. При копировании операция прервалась с сообщением, больше места нет. Но 10 млн файлов даже если они все по 4 Кб (на самом деле в среднем они же меньше) — это всего лишь 40 Gb. То есть, на разделе должно быть ещё свободно больше половины пространства. И действительно, команда df показывает, что больше половины раздела свободно. Как так получается и что можно сделать, чтобы всё-таки поместить в раздел все файлы?

3. Как определить владельца файла?

4. Почему нельзя вынимать/удалять флэшку до её демонтирования?

5. «Корзина для удалённых файлов» - что это, кто создаёт, как это работает?

6. Можно ли считать каталог «lost+found» корзиной для удалённых файлов?

ЛЕКЦИЯ 5. ФАЙЛОВАЯ СИСТЕМА EXT-2/3/4

5.1. Общие сведения

Преданья старины глубокой сообщают нам, что на заре развития Linux использовала файловую систему ОС Minix. Она была довольно стабильна, но оставалась 16-разрядной и, как следствие, имела жёсткое ограничение в 64 Мегабайта на раздел и 14 символов на максимальную длину имени файла. Эти и другие ограничения послужили стимулом к разработке «расширенной файловой системы» (*Extended File System*), которая была представлена в апреле 1992 года.

Ext расширила ограничения на размер файла до 2 гигабайт и установила предельную длину имени файла в 255 байт, как в unix-ах.

Тем не менее, оставались ещё проблемы: не было поддержки раздельного доступа, временных меток модификации данных и др. Это послужило причиной для создания следующей версии расширенной файловой системы ext2 (*Second Extended File System*), разработанной в январе 1993 года. В ext2 были также реализованы соответствующие стандарту POSIX списки контроля доступа ACL и расширенные атрибуты файлов.

Файловая система ext2 создавалась по образу и подобию хорошо отработанной и очень качественной файловой системы UFS. Но в силу того, что в начале 90-х linux почти всегда использовалась на ПЭВМ, а ПЭВМ в те годы были весьма «слабыми» (вы даже не представляете, какими они были слабыми!). Всё таки, 286, 386 и максимум 486 процессоры — это весчь! Поэтому реализовать в файловой системе ext2 всю мощь, красоту и отработанность UFS не было никакой возможности.

Граф, описывающий иерархию каталогов файловой системы ext2, представляет собой ациклический граф, а не дерево. Ациклический — означает, что связи между вершинами графа (каталогами и файлами) — однонаправленные. Причина такой организации заключается в том, что в файловой системе определено понятие «жёсткой ссылки»: каждое имя файла — это жёсткая ссылка и они могут быть расположены в любом месте файловой системы.

В файловой системе определены:

- имя файла («простое имя файла») — символьное имя длиной до 255 байт, в имени не могут присутствовать символы NULL, «/» (слэшь) и «;» (точка с запятой»);

- относительное имя - цепочка простых символьных имен всех каталогов, через которые проходит путь от текущего места до данного файла;

- абсолютное имя файла (иногда говорят «полное имя») - цепочка простых символьных имен всех каталогов, через которые проходит путь от **корня файловой системы** до данного файла; абсолютное имя файла всегда начинается с символа «/» (слэшь).

Основные атрибуты файла (отображаемые командой ls, берутся из строки индексной таблицы, описывающей файл):

- тип файла,
- режим доступа к файлу,
- владелец («хозяин») файла,
- группа хозяина файла,
- количество имён у файла,
- информация о разрешённых операциях (ACL),
- четыре временных штампа: время создания, дата последнего доступа, дата последнего изменения и время последнего удаления файла,
- размер файла,
- адреса занимаемых блоков,
- индекс файла — номер строки индексной таблицы, в которой все эти атрибуты хранятся,
- а также другие атрибуты, командой ls не отображаемые.

Отдельно, в каталоге, хранится имя файла.

5.2. Структура дискового раздела с файловой системой ext2

Структура раздела с файловой системой ext2 представлена на рис. 46. Всё пространство раздела диска разбивается на блоки фиксированного размера, кратные размеру сектора: 1024, 2048, 4096 или 8192 байт. Размер блока указывается при создании файловой системы на разделе диска. Меньший размер блока позволяет сэкономить место на жёстком диске, но также ограничивает максимальный размер файловой системы. Все блоки имеют порядковые номера. С целью уменьшения фрагментации и количества перемещений головок жёсткого диска при чтении больших массивов данных блоки объединяются в группы блоков.

Суперблок — основной элемент файловой системы ext2. Он содержит общую информацию о файловой системе:

- общее число блоков в файловой системе,
- общее число строк индексной таблицы, строка индексной таблицы - «индексный дескриптор»,
- число свободных блоков в файловой системе,
- число свободных индексных дескрипторов — строк в индексной таблице,
- размер блока файловой системы,
- количество блоков в группе блоков,
- количество индексных дескрипторов в группе блоков,
- размер индексного дескриптора - длина строки индексной таблицы,
- идентификатор файловой системы (магическое число 0xEF53 для семейства файловых систем ext),
- дата последней проверки файловой системы,
- количество произведённых монтирований,
- флаг состояния файловой системы — флаг «чистоты».

Суперблок (первая копия) находится в 0-ой группе блоков в 1024 байтах от начала раздела (первые 1024 байта зарезервированы для загрузчика операционной системы). Другие копии находятся в группах блоков 1, 3, 5, 7, 9, 25, 27, 49, 81, 125, 243, 343, 625, 729 и т. д.

В следующем блоке после суперблока располагается **глобальная дескрипторная таблица** (таблица описателей групп блоков) — описание групп блоков, представляющее собой массив (то есть, двумерная таблица с количеством строк, равным количеству групп блоков), содержащий общую информацию обо всех группах блоков раздела. Как и суперблок, она имеет копии в группах блоков. Её структура (не все поля показаны):

struct ext2_group_desc:
__u32 bg_block_bitmap – битовая карта занятости блоков группы;
__u32 bg_inode_bitmap – битовая карта занятости inode группы;
__u32 bg_inode_table – адрес таблицы inode группы.

Далее начинается первая группа блоков.

В первом блоке каждой группы блоков содержится **битовая карта блоков** — структура, каждый бит которой показывает, отведён ли соответствующий ему блок какому-либо файлу. Если бит равен 1, то блок занят.

В следующем блоке содержится **битовая карта индексных дескрипторов**, которая показывает, какие именно индексные дескрипторы (строки индексной таблицы)заняты, а какие нет.

В следующих нескольких блоках содержится **индексная таблица** (таблица индексных дескрипторов, или таблица inode), точнее 1/n-ая часть её, где n — количество групп блоков. То есть, вся индексная таблица файловой системы делится поровну между всеми группами блоков и каждая часть помещается в соответствующую группу блоков, сразу после битовых карт. Её структура показана в пункте 5.4.

Ядро Linux пытается равномерно распределить каталоги по группам блоков, а файлы, соответственно, старается по возможности переместить в одну группу с родительским каталогом.

Все оставшееся место отводится для хранения файлов.

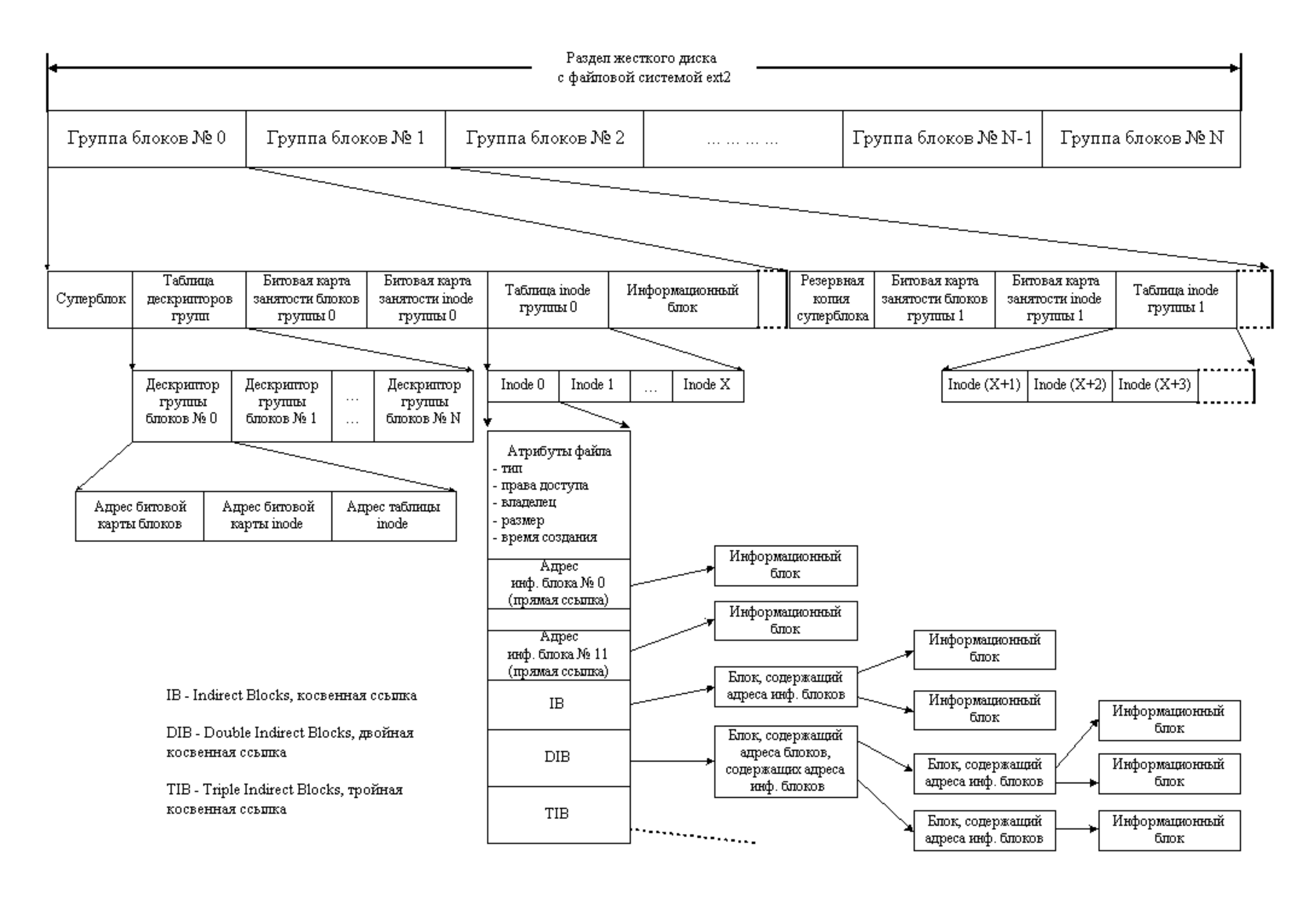


Рис. 46. Структура файловой системы ext-2.

Размер блока файловой системы указывается при форматировании раздела. От его размера косвенно зависят размер группы блоков, количество групп блоков на разделе, размер индексной таблицы и максимальный размер файла.

5.3. Каталог в файловой системе ext2

Файл типа каталог, имеющий структуру таблицы, но хранящийся как список, показан в таблице 8.

Таблица 8. Каталог в ext2.

I - Индекс файла, порядковый номер строки индексной таблицы, в которой содержатся атрибуты файла, 4 байта	L - Общая длина строки каталога — длина всех полей строки в байтах, 2 байта	l — длина имени файла в байтах, 1 байт	Имя файла, до 255 байт

. . .	. . .	. . .	. . . .
N = {от 2 до 4 млрд}	15	8	file.txt
. . . .	. . .	. . .	. . .

Структура записи каталога struct ext2_dir_entry_2 показана на рис. 47.

```
#define EXT2_NAME_LEN 255
__u32 inode                    - номер inode-а файла
__u16 rec_len                  - длина записи каталога
__u8 name_len                  - длина имени файла
char name [EXT2_NAME_LEN]     - имя файла
```

Рис. 47. Описание каталога в исходниках ext2.

Отдельная запись в каталоге не может пересекать границу блока (то есть, должна быть расположена целиком внутри одного блока). Поэтому, если очередная запись не помещается целиком в данном блоке, она переносится в следующий блок, а предыдущая запись продолжается таким образом, чтобы она заполнила блок до конца.

В ext4 в этой таблице может быть ещё одна колонка — тип файла, 1 (или 2) байта.

5.4. Индексная таблица

Структура индексной таблицы (не все поля показаны):

struct ext2_inode:
__u16 i_mode – тип файла и права доступа к нему. Тип файла определяют биты 12-15 этого поля:

0xA000 – символическая ссылка;
0x8000 – обычный файл;
0x6000 – файл блочного устройства;
0x4000 – каталог;
0x2000 – файл символьного устройства;
0x1000 – канал FIFO.

__u32 i_size – размер в байтах;
__u32 i_atime – время последнего доступа к файлу;
__u32 i_ctime – время создания файла;
__u32 i_mtime – время последней модификации;
__u32 i_blocks – количество блоков, занимаемых файлом;
__u32 i_block[EXT2_N_BLOCKS] – адреса информационных блоков (включая все косвенные ссылки).

Константа EXT2_N_BLOCKS определена в файле так:

```
/*
 * Constants relative to the data blocks
 */
#define EXT2_NDIR_BLOCKS 12
#define EXT2_IND_BLOCK     EXT2_NDIR_BLOCKS
```

```
#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS   (EXT2_TIND_BLOCK + 1)
```

Несколько первых строк индексной таблицы зарезервированы файловой системой, их перечень содержится в заголовочном файле (хидере):

```
/*
 * Special inode numbers
 */
#define EXT2_BAD_INO 1 /* Bad blocks inode */
#define EXT2_ROOT_IN 2 /* Root inode */
#define EXT2_ACL_IDX_IN 3 /* ACL inode */
#define EXT2_ACL_DATA_INO 4 /* ACL inode */
#define EXT2_BOOT_LOADER_INO 5 /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO 6 /* Undelete directory inode */
```

5.5. Алгоритмы файловой системы ext2

При чтении файла, определив порядковый номер inode файла из каталога, система вычисляет номер группы, в которой этот inode расположен, и его позицию в таблице inode этой группы. Считав из этой позиции inode, операционная система получает полную информацию о файле, включая адреса блоков, в которых хранится содержимое файла.

Пример получения содержимого файла test.file, находящегося в корневом каталоге. Для чтения файла /test.file необходимо:

- в массиве записей корневого каталога найти запись об этом файле;
- извлечь порядковый номер inode файла, вычислить номер группы, в которой этот inode расположен;
- из дескриптора данной группы извлечь адрес таблицы inode группы;
- вычислить позицию inode в этой таблице;
- считать inode файла;
- из inode извлечь адреса информационных блоков и осуществить чтение информации, находящейся в этих блоках.

На рис. 48 подробно показаны этапы чтения файла /test.file.

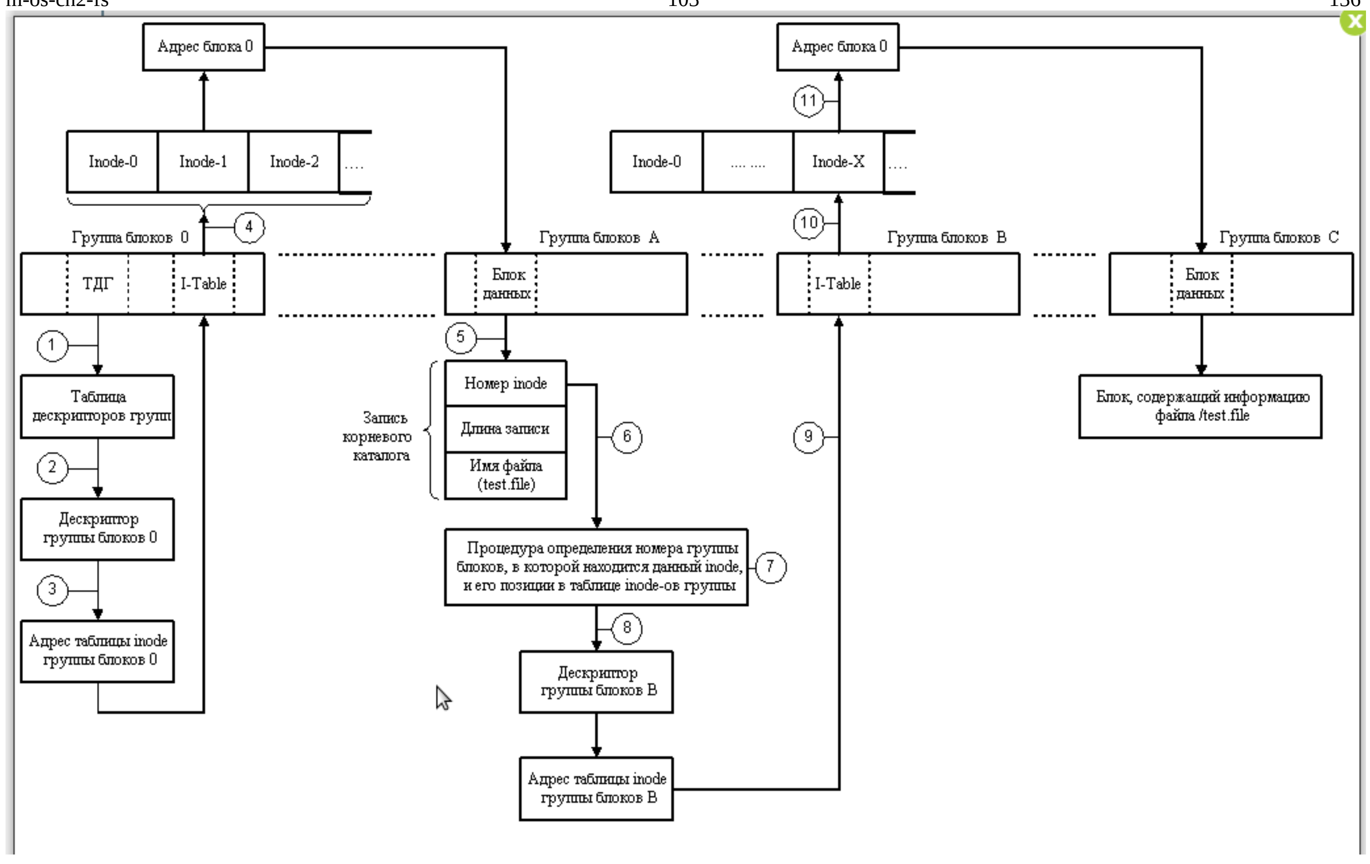


Рис. 48. Алгоритм чтения файла в файловой системе ext.

Этапы 1-6 – чтение корневого каталога:

1. Из группы блоков 0 считывается таблица дескрипторов групп.
2. Из таблицы дескрипторов групп извлекается дескриптор группы блоков 0 и из него считывается адрес таблицы inode группы 0.
3. Из группы блоков 0 считывается таблица inode.
4. Порядковый номер inode корневого каталога фиксирован и равен 2, поэтому из таблицы inode группы 0 считывается второй элемент, который содержит адрес блока с содержимым корневого каталога. Предположим, что этот блок расположен в группе блоков A.
5. Из группы блоков A считывается блок, содержащий записи корневого каталога.
6. Выполняется поиск записи с именем «test.file». Если такая запись найдена, из нее извлекается порядковый номер inode файла «test.file».

Определив номер inode, можно получить доступ к информационным блокам файла (этапы 7-11):

7. Вычисляется номер группы, в которой находится данный inode, и его позицию в таблице inode группы (предположим, что номер группы равен B, а позиция в таблице – X).
8. Из таблицы дескрипторов групп извлекаем дескриптор группы блоков B, и из него считывается адрес таблицы inode этой группы блоков.
9. Из группы блоков B считывается таблица inode.
10. Из таблицы inode группы блоков B считывается inode, находящийся в позиции X.
11. Из считанного inode извлекаются адреса блока с содержимым файла /test.file и выполняется чтение информации из блока с указанным адресом.

Следует учесть, что созданный файл не сразу записывается на диск, а сначала попадает в дисковый буфер. Попытка сразу же получить содержимое файла по вышеприведенному алгоритму ни к чему не приведет, так как информация об этом файле физически на диске отсутствует. Необходимо «заставить» систему записать дисковый буфер на диск. Самый простой способ сделать это – подождать несколько минут. Либо, если срочно надо, дать команду sync.

5.6. Файловая система ext3

Существенным недостатком файловой системы ext2 была не очень высокая надёжность хранения файлов. Конечно, для ПЭВМ начала-середины 90-х и их области применения надёжности вполне хватало. Однако к концу 90-х, с появлением новых процессоров и гигабайтных винчестеров, ПЭВМ начали широко применяться в «реальной экономике». Соответственно, повысились требования к надёжности хранения информации. А у ext2 был серьёзный недостаток: при заполненности раздела на 70 и более процентов надёжность файловой системы резко падала — при внезапном выключении (или нажатии Reset) легко можно было недосчитаться файлов и даже регулярное выполнение fsck не очень помогало.

Третья расширенная файловая система (ext3) была серьезным продвижением вперед для файловых систем Linux, хотя и уступала по производительности некоторым своим конкурентам. В ext3 появилась концепция журналирования, призванная улучшить надежность файловой системы в случае внезапной остановки. И хотя конкурирующие файловые системы (такие как XFS от Silicon Graphics и журналируемая файловая система IBM - JFS) обладали лучшей производительностью, преимуществом ext3 была возможность обновления "налету" системы, использующей ext2, до ext3. Ext3 была разработана **Стефаном Твиди** и появилась в ноябре 2001 года.

5.7. Файловая система ext4

А сейчас, когда ПЭВМ сравнились по мощности и даже превзошли мейнфреймы 90-х, соответственно, требования к возможностям файловых систем ПЭВМ уже ничем не отличаются от таковых для суперкомпьютеров. Тем более, что 95 % всех суперкомпьютеров в мире в настоящее время используют linux — да-да, ту самую ОС для ПЭВМ [16].

Поэтому маленькие мальчики теперь играют по-взрослому и появилась четвертая расширенная файловая система (ext4). В ext4 сделано несколько новых улучшений производительности и надежности. Самым заметным, пожалуй, является то, что ext4 поддерживает файловые системы до одного экзабайта. Ext4 была реализована командой программистов под руководством **Теодора Цо** (разработчика, сопровождающего ext3). Она впервые появилась в ядре версии 2.6.19 и стала стабильной в ядре 2.6.28, вышедшем в декабре 2008 года. Ext4 позаимствовала множество полезных идей из ряда конкурирующих файловых систем. К примеру, управление блоками на основе экстентов было ранее реализовано в JFS. Другая новая возможность ext4, связанная с управлением блоками - отложенное выделение, была ранее реализована в XFS и ZFS от Sun Microsystems.

В новой файловой системе ext4 находим множество разнообразных улучшений и инноваций. Добавлена новая функциональность, сделаны улучшения в масштабируемости (позволившие преодолеть ограничения системы ext3), надежности (по отношению к сбоям) и, безусловно, в производительности.

5.7.1. Функциональность

Ext4 включает в себя большое количество новой функциональности, но самой важной является поддержка прямой и обратной совместимости с системой ext3 и улучшения в метках

времени, сделанные с расчетом на будущие более производительные Linux-системы.

Для безболезненного перехода от ext3 к ext4 новая файловая система была спроектирована так, чтобы иметь прямую и обратную (в некоторой степени) совместимость с ext3. Прямая совместимость означает, что можно монтировать файловую систему ext3 как систему ext4. Чтобы пользоваться всеми преимуществами ext4 в полной мере, необходимо выполнить процедуру миграции для конвертации системы в формат ext4. Обратная совместимость означает, что можно монтировать файловую систему ext4 как систему ext3, но только если система ext4 не использует экстенты.

В дополнение к возможностям совместимости также можно осуществлять постепенный переход с ext3 на ext4. То есть, старые файлы могут оставаться в формате ext3, в то время как новые файлы (или скопированные старые файлы) будут располагаться уже в структурах данных новой системы ext4. Таким образом, можно переходить на систему ext4 в процессе работы.

5.7.2. Увеличение точности и диапазона временных меток

Как ни удивительно, но раньше в расширенных файловых системах использовались временные метки с точностью до секунды. Для многих целей такая точность была достаточна, но по мере увеличения скоростей процессоров и появления многоядерных процессоров, а также распространения Linux в таких областях, как высокопроизводительные вычисления, секундной точности временных меток стало не хватать. С расчетом на будущее в файловой системе ext4 точность временных меток была увеличена до одной наносекунды путем добавления еще одного (младшего) значащего байта. Также на два байта был расширен временной интервал, что увеличило возможное время жизни системы на 500 лет, до 25 апреля 2514 года

5.7.3. Масштабируемость

Одним из самых важных аспектов развития файловых систем является возможность масштабировать их в соответствии с ростом предъявляемых к ним требований. Ext4 достигает этого несколькими способами, выходя за пределы ограничений ext3 и реализуя новые возможности управления метаданными файловой системы.

Расширение лимитов файловой системы. Одним из самых заметных отличий ext4 от ext3 является поддержка увеличенных размеров томов, файлов и поддиректорий. Ext4 поддерживает файловые системы размером до одного экзабайта (1000 петабайт). Хотя по сегодняшним стандартам это громадная цифра, потребление места на устройствах хранения увеличивается, так что, несомненно, ext4 была разработана с расчетом на будущее. Файлы в ext4 могут достигать размера 16 ТБ (при блоках размером 4 КБ), что в восемь раз больше, чем в ext3. И наконец, глубина поддиректорий в ext4 была увеличена с 32 000 до фактически бесконечной. Это может показаться избыточным, но тут надо принимать во внимание возможную иерархию файловой системы размером в экзабайт. Также было оптимизировано индексирование директорий, которое теперь использует хэширующую структуру наподобие B-дерева. Поэтому, несмотря на гораздо больший размер, поиск в ext4 работает очень быстро.

Экстенты. Одним из главных недостатков системы ext3 был ее метод выделения места на дисках. Дисковые ресурсы для файлов выделялись с помощью битовых карт свободного места - способа, не выделяющегося ни скоростью, ни масштабируемостью. Формат, применяемый в ext3, очень эффективен для маленьких файлов, но ужасно неэффективен для больших. Поэтому для улучшения выделения ресурсов и поддержки более эффективной структуры хранения данных в ext4 вместо битовых карт применяются экстенты. Экстент - это способ представления непрерывной последовательности блоков памяти. При использовании экстентов сокращается количество метаданных, так как вместо информации о том, где находится каждый блок памяти,

экстенты содержат информацию о том, где находится большой список непрерывных блоков памяти.

В ext4 для эффективного представления маленьких файлов в экстентах применяется уровневый подход, а для эффективного представления больших файлов применяются деревья экстентов. Например, один индексный дескриптор в ext4 имеет достаточно места, чтобы ссылаться на четыре экстента (каждый из которых представляет множество последовательных блоков). Для больших (в том числе фрагментированных) файлов дескриптор может содержать ссылки на другие индексные дескрипторы, каждый из которых может указывать на концевой узел (указывающий на экстенты). Такое дерево экстентов постоянной глубины предоставляет мощный механизм представления больших, потенциально фрагментированных файлов. Также узлы имеют механизмы самопроверки для защиты от повреждений файловой системы.

5.7.4. Производительность

Одним из самых важных параметров, используемых при оценке новых файловых систем, является их фундаментальная производительность. Это одна из наиболее сложных областей, которая по мере роста размеров файловых систем и высоких требований к их надежности может серьезно пострадать. И все же ext4, вместе с повышением масштабируемости и надежности, имеет ряд улучшений, связанных с производительностью.

Предварительное выделение на файловом уровне. Некоторые приложения, например базы данных или потоковое вещание, рассчитывают, что их файлы будут храниться в непрерывных блоках (чтобы использовать оптимизацию при последовательном чтении данных с дисков, а также минимизировать количество команд Read в расчете на блок данных). Хотя сегменты непрерывных блоков можно получить с помощью экстентов, есть и другой, более грубый метод: предварительно выделять очень большие сегменты непрерывных блоков желаемого размера (как это было реализовано раньше в XFS). Ext4 делает это с помощью нового системного вызова, который осуществляет предварительное выделение и инициализацию файла заданного размера. Далее можно записывать необходимые данные и читать их с известной производительностью посредством операций Read.

Отложенное выделение блоков памяти. Другим приемом оптимизации, основанным на размере файлов, является отложенное выделение. Этот прием оптимизации производительности откладывает выделение физического блока памяти до момента, когда данные должны быть фактически записаны на диск. Суть этой оптимизации в том, что откладывание выделения физических блоков памяти до того, пока они действительно будут записаны, позволяет выделять больше последовательных блоков. Это похоже на предварительное выделение, за исключением того, что эту задачу система выполняет автоматически. Но если размер файла известен заранее, лучше применять предварительное выделение.

Выделение блоков памяти группами. И последняя оптимизация, также связанная с последовательными блоками, - это групповое выделение блоков в ext4. В ext3 каждый блок выделяется по отдельности. Поэтому иногда получалось, что для последовательных данных выделенные блоки располагались не последовательно. В ext4 эта проблема решена за счет того, что выделение группы блоков происходит за один раз, поэтому фрагментирование маловероятно. Здесь, как и при предыдущей технике оптимизации, связанные данные хранятся на диске вместе, что в свою очередь позволяет оптимизировать их чтение.

Другим аспектом группового выделения блоков является объем работы, необходимой для выделения блоков. Напомним, что в ext3 выделение осуществляется по одному блоку за раз. Выделение блоков группами требует гораздо меньшего количества вызовов, что ускоряет выделение блоков.

5.7.5. Надежность

При увеличении размеров файловых систем до уровней, поддерживаемых ext4, неизбежно встает проблема повышения надежности. Для ее решения в ext4 предусмотрено множество механизмов самозащиты и самовосстановления.

Контрольная сумма журнала файловой системы. Как и ext3, ext4 является журналируемой файловой системой. Журналирование - это процесс записи изменений, происходящих в файловой системе в журнал (специально выделенный журнальный файл с круговой записью, занимающий непрерывную область на диске). Фактические изменения на физическом устройстве делаются из файла журнала. Это позволяет производить изменения более надежным образом и гарантировать целостность данных даже в случае краха системы или сбоя питания во время выполнения операции. В результате снижается вероятность повреждения файловой системы.

Но даже с применением журналирования повреждения системы возможны, если в журнал попадут ошибочные записи. Для борьбы с этим в ext4 реализована проверка контрольных сумм записей журнала, чтобы гарантировать, что в нижележащую файловую систему будут внесены правильные изменения.

В зависимости от нужд пользователя ext4 может работать в разных режимах журналирования. Например, ext4 поддерживает режим обратной записи (writeback - в журнал заносятся только метаданные), режим упорядочивания (ordered - метаданные заносятся в журнал, но только после записи самих данных), а также самый надежный - журнальный режим (journal - в журнал заносятся как данные, так и метаданные). Заметьте, что хотя журнальный режим - это лучший способ гарантировать целостность файловой системы, в то же время это и самый медленный режим, так как в нем через журнал проходят все данные.

Дефрагментация "на лету". Хотя ext4 включает в себя возможности уменьшения фрагментации внутри файловой системы (экстенты для выделения последовательных блоков), все же при длительной жизни файловой системы некоторая фрагментация неизбежна. Поэтому для улучшения производительности существует инструмент, который на лету дефрагментирует как файловую систему, так и отдельные файлы. Дефрагментатор - это простой инструмент, который копирует (фрагментированные) файлы в новый дескриптор ext4, указывающий на непрерывные экстенты.

Другим результатом дефрагментации на лету является уменьшение времени проверки файловой системы. (fsck). Ext4 помечает неиспользуемые группы блоков в таблице индексных дескрипторов, что позволяет процессу (fsck) полностью их пропускать и ускоряет тем самым процедуру проверки. Поэтому, когда операционная система решит проверить файловую систему после внутреннего повреждения (которые неизбежно будут происходить по мере увеличения размера файловой системы и ее распределенности), благодаря архитектуре ext4 это можно будет сделать быстро и надежно.

5.7.6. Что дальше?

Расширенная файловая система в Linux имеет долгую и богатую историю - от первого появления ext в 1992 году до выхода ext4 в 2008. Это была первая файловая система, разработанная специально для Linux, и она зарекомендовала себя как одна из самых эффективных, стабильных и мощных файловых систем. Ext4 развивалась вместе с прогрессом в исследовании файловых систем, заимствуя идеи из других новых файловых систем (таких как XFS, JFS, Reiser, а также IRON-технологии отказоустойчивых файловых систем). Хотя сейчас слишком рано говорить о том, какой будет ext5, ясно, что она будет развиваться с учетом потребностей Linux-систем корпоративного уровня.

Тем не менее, есть мнение, что Ext4 – тупиковая ветвь эволюции, развиваться она не будет. Из архитектуры Ext больше ничего нельзя выжать, да и разработчики сконцентрировали усилия на **Btrfs. Д**ело в том, что linux очень быстро развивается, активно используется корпоративным сектором экономики, задействована в новейших исследованиях в науке, используется как основа для разработки критических систем (высокой надёжности и доступности 24x7). И это сказывается. Но в то же время говорят, что это мнение неверное и в ближайшие годы можно ожидать появления ext5..

Вопросы на «засыпку»

1. Можно ли файловую систему ext2 смонтировать как файловую систему ext4?
2. Существует ли файловая система, в которой имя файла хранится в жёсткой ссылке?
3. Какие символы нельзя использовать в именах файлов в файловой системе EXT4?
4. Как используются фрагменты блока в файловых системах ext2/3/4?
5. Если у вас в компьютере установлены винчестер (с двумя разделами) и один SSD (с одним разделом), то сколько у вас корневых каталогов?
6. В пункте 5.3. написано: «Отдельная запись в каталоге не может пересекать границу блока (то есть, должна быть расположена целиком внутри одного блока). Поэтому, если очередная запись не помещается целиком в данном блоке, она переносится в следующий блок, а предыдущая запись продолжается таким образом, чтобы она заполнила блок до конца.»

   Как это реализуется?
7.

ЛЕКЦИЯ 6. ФАЙЛОВЫЕ СИСТЕМЫ FAT И NTFS

6.1. Файловая система FAT

6.1.1. Структура

Разработана Биллом Гейтсом и Марком МакДональдом в 1976—1977 годах. Использовалась в качестве основной файловой системы в операционных системах семейств MS-DOS и Windows 9x. Существует четыре версии FAT — **FAT12**, **FAT16**, **FAT32** (см. рис. 49) и **exFAT (FAT64).**

В начале раздела идут «зарезервированные» сектора — от одного до трёх. В первом из них (в нулевом по LBA) находится «Блок Параметров BIOS» (BPB - BIOS parameter block, см. таблицу 9), размером 79 байт - аналог суперблока, и, ниже располагается вторичный загрузчик. В таблице 9 показан полный BPB, а бывает и неполные, которые в более ранних версиях ОС. Смещение BPB в секторе на 11 байт обусловлено тем, что в первых байтах идёт команда jmp на начало вторичного загрузчика: первичный загрузчик грузит вторичный в память и передаёт управление на начало сектора, как раз на эту команда. Если вторичный загрузчик не помещается в первом секторе, то задействуются второй и третий зарезервированные сектора. Каждый зарезервированный сектор завершается сигнатурой 0xAA55 в последних двух байтах.

В поле BPB со смещением 0x25 расположен адрес структуры FSInfo с дополнительной информацией о файловой системе (см. таблицу 10). Обычно она располагается с следующем, первом по LBA, секторе.

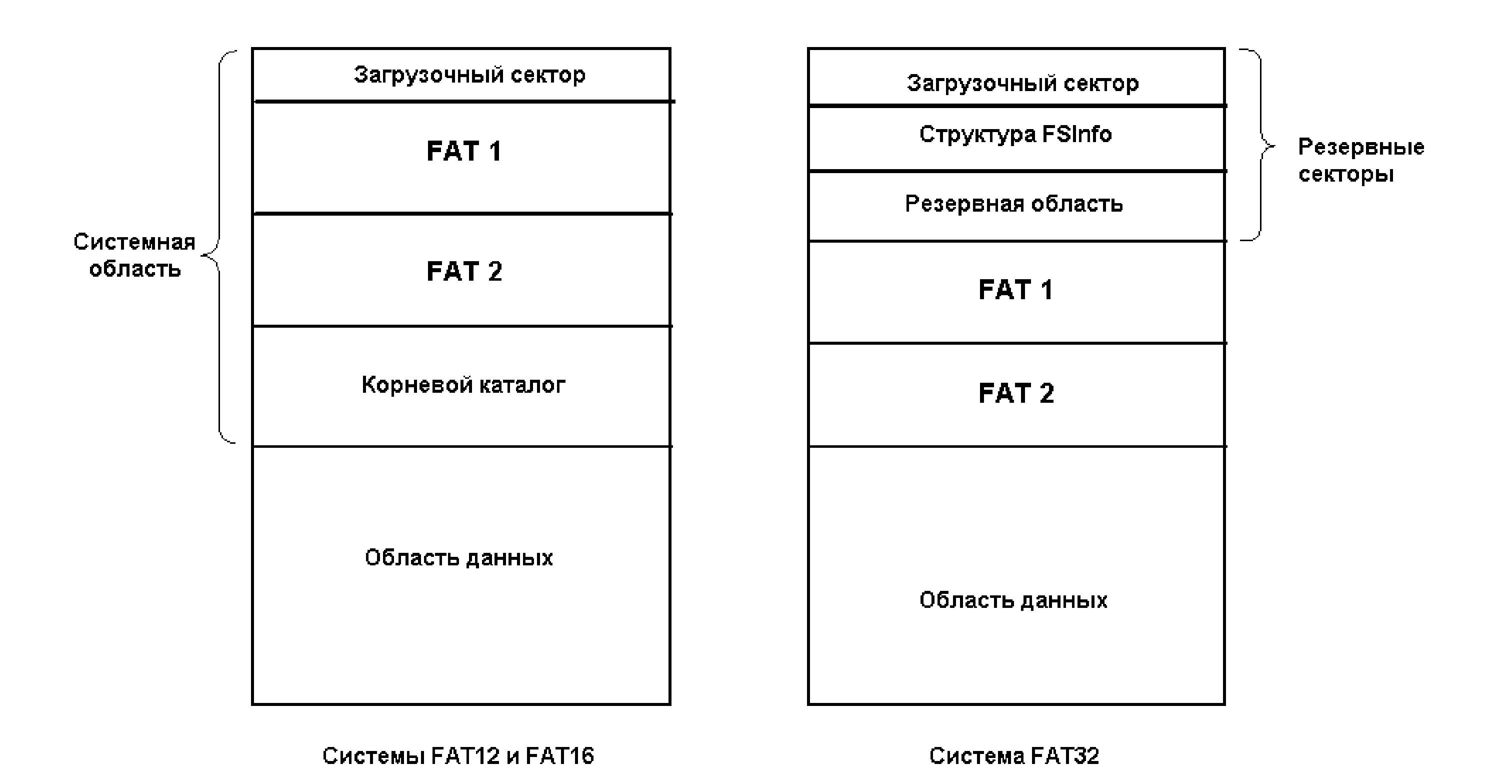


Рис. 49. Структура раздела с файловой системой FAT. То есть, в FAT32 корневой каталог может находиться не сразу за таблицами FAT, а подальше, где-то в области данных. Его размещение можно указать при форматировании раздела.

Таблица 9. BPB.

Смещение в	Смещение	Длина	Описание

секторе	в BPB		
0x00B	0x00	25 байт	DOS 3.31 BPB
0x024	0x19	4 байта (dword)	Размер FAT в логических секторах
0x028	0x1D	2 байта (word)	Двойное поле флагов (два одинаковых байтовых поля)
0x02A	0x1F	2 байта (word)	Версия
0x02C	0x21	4 байта (dword)	Корневой каталог в кластерах
0x030	0x25	2 байта (word)	Расположение FSI-структуры (дополнительный блок информации о файловой системе)
0x032	0x27	2 байта (word)	Расположение резервных секторов
0x034	0x29	12 байт	Зарезервировано (имя файла загрузки)
0x040	0x35	1 байт	Номер физического диска
0x041	0x36	1 байт	Флаги
0x042	0x37	1 байт	Расширенная загрузочная запись
0x043	0x38	4 байта (dword)	Серийный номер тома
0x047	0x3C	11 байт	Метка тома
0x052	0x47	8 байт	Тип файловой системы

Таблица 10. FSInfo (сектор 1).

Смещение	Размер	Описание	Обязательное
0	4	Последовательность байт **0x41615252**	Нет
4	480	Резерв	Нет
484	4	Последовательность байт **0x6147272**	Нет
488	4	Сколько свободных кластеров	Нет
492	4	Следующий свободный кластер	Нет
496	12	Резерв	Нет
508	4	Последовательность байт **0xAA550000**	Нет

Определение типа файловой системы FAT (то есть выбор между FAT12, FAT16 и FAT32) производится ОС по количеству кластеров в разделе:

если меньше 4085 — FAT12,
если 4085 ÷ 65 524 — FAT16,
если больше 65 52 — FAT32.

Далее идут одна за другой две таблицы FAT, в которых строк чуть-чуть больше, чем кластеров на разделе.

Далее идёт корневой каталог (для FAT12 и FAT16).

Далее обычные файлы и каталоги и где-то среди них — корневой каталог (для FAT32).

6.1.2. Таблицы FAT

Таблица FAT - одномерная таблица, то есть, таблица из одной колонки. Каждое поле — два байта для FAT-12/16, или четыре байта — для FAT-32. Каждая строка соответствует кластеру в файловой системе — взаимно однозначное соответствие.

Если в строке таблицы 0 (нуль), то кластер свободен.

Если кластер занят файлом и не является последним кластером файла — указатель содержит номер следующего кластера файла.

Если код 0x0FF8 (для FAT12), или 0x0FFF8 (для FAT16) или 0x0FFFFFFF8 (для FAT32), то это кластер конца файла; но иногда используются коды 0xFFF, 0x0FFFF, 0x0FFFFFFFF.

Если код 0xFF7, или 0x0FFF7 или 0x0FFFFFFF7, то сбойный кластер.

Значение размера таблицы FAT по BPB, то есть BPB_FATSz16/32, может превышать реальное, так что в конце каждой таблицы FAT могут находиться сектора, не соответствующие никаким реальным кластерам данных. При форматировании эти сектора обнуляются, а в процессе функционирования тома никак не используются. Поэтому действительный адрес последнего сектора таблицы FAT, содержащего указатели на реальные кластеры тома, всегда должен рассчитываться из общего количества кластеров области данных, а не из поля BPB_FATSz16/32. Кроме того, последний сектор, занятый таблицей FAT, вовсе не обязательно весь занят ею — в этом случае избыточное пространство сектора также не используется и забивается нулями при форматировании тома.

При создании файла Windows ищет в таблице FAT первую строку с нулём (или в первом байте каталожной записи), соответствующую свободному кластеру. Он и выделяется создаваемому файлу, как начальный. При этом не проверяется, «влезет» ли весь файл в этот кластер, или файл много длиннее. Если длиннее, а следующий за выделенным кластер некоторым образом занят (возможны три случая: занят файлом, bad, занят удалённым файлом), то Windows ищет следующую строку с нулём. И т. д. Такой подход к выделению места для файлов приводит к усиленной фрагментации файлов.

Если больше нет строк с нулём, то тогда для размещения файла берётся первая строка с кодом 0xE5 (или 0x05) в первом байте каталожной записи (признак удалённого файла), то есть, начинают использоваться кластеры, занятые удалёнными файлами.

6.1.3. Корневой каталог

Единственным обязательно присутствующим каталогом в разделе FAT является корневой каталог. В FAT12/FAT16 корневой каталог имеет фиксированный размер в секторах, который вычисляется из значения BPB_RootEntCnt, и следует на диске непосредственно после таблицы FAT.

В FAT32 корневой каталог, как и любой другой, имеет переменный размер и является цепочкой кластеров. Номер первого кластера корневого каталога отражается BPB_RootClus. Корневой каталог имеет следующие отличия от других каталогов тома FAT:

- у него нет меток даты и времени;
- нет собственного имени (кроме «\»);
- он не содержит файлов с именами «.» и «..» (см. далее);
- является единственным каталогом, в котором может располагаться файл метки тома.

6.1.4. Каталоги и файловые записи в каталогах

Каталог в файловых системах FAT-12/16/32 (см. рис. 50) является обычным файлом, помеченным специальным атрибутом в поле DIR_Attr — значение 0x10. Формат этого файла — таблица из 12 колонок. Каждая строка таблицы имеет фиксированную длину 32 байта - это файловые записи (записи каталога).

Внимание! FAT-32 называется FAT именно 32 вовсе не потому, что длина строки каталога равна 32 байтам.

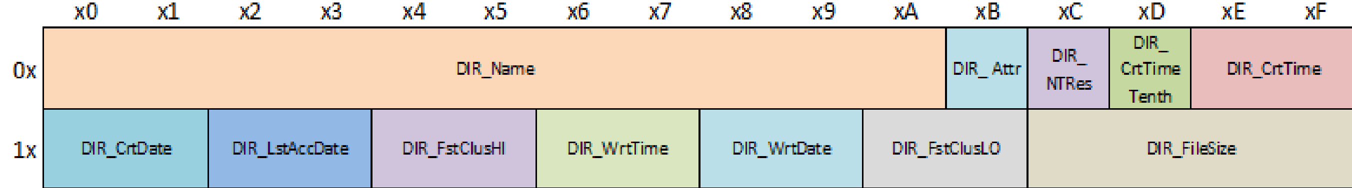


Рис. 50. Строка каталога файловой системы FAT-32 — SFN-запись.

Файловая запись FAT (SFN-запись — Short File Name) состоит из следующих полей:

1) DIR_Name - 11-байтовое поле по относительному адресу 0, содержит короткое имя файла (в рамках стандарта 8.3; для точки здесь места нет, точка добавляется на уровне операционной системы).
2) DIR_Attr - байт атрибутов файла.
3) DIR_NTRes - байт используется в Windows NT, зарезервирован.
4) DIR_CrtTimeTenth - байт счётчика десятков миллисекунд времени создания файла, допустимы значения 0–199. Поле обычно игнорируется.
5) DIR_CrtTime - 2 байта определяют время создания файла с точностью до 2 секунд.
6) DIR_CrtDate - 2 байта даты создания файла.
7) DIR_LstAccDate - 2 байта даты последнего доступа к файлу (то есть последнего чтения или записи — в последнем случае приравнивается DIR_WrtDate). Аналогичное поле для времени не предусмотрено.
8) DIR_FstClusHI - 2 байта номера первого кластера файла (старшее слово, в файловых системах FAT12/FAT16 равен нулю).
9) DIR_WrtTime - 2 байта определяют время последней записи (модификации) файла, например его создания.
10) DIR_WrtDate - 2 байта даты последней записи (модификации) файла, в том числе создания.
11) DIR_FstClusLO - 2 байта номера первого кластера файла (младшее слово, используется в FAT-12/16/32).
12) DIR_FileSize — 4байта, содержащие значение размера файла в байтах. Фундаментальное ограничение FAT32 — максимально допустимое значение размера файла составляет 0xFFFFFFFF (то есть 4 Гбайт минус 1 байт).

Если первый байт записи FAT (то есть DIR_Name[0]) содержит 0xE5 или 0x05, это значит, что запись свободна (соответствующий файл был удалён; остальная часть SFN-записи не меняется — только первый байт). Ноль в DIR_Name[0] означает, что свободна не только эта запись, но и все следующие записи каталога; Windows не анализирует остаток каталога после обнулённой записи. Кстати, она может быть не совсем обнулена, а только первый байт.

На использование ASCII-символов в коротком имени накладывается ряд ограничений:

- нельзя использовать символы с кодами меньше 0x20 (за исключением кода 0x05 в первом байте короткого имени);
- нельзя использовать символы с кодами 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, 0x7C;
- нельзя использовать символ пробела (0x20) в первом байте имени.

Если файловая система поддерживает длинные имена (FAT-32/vFAT/exFAT до 255 символов), то для такого файла всё равно заводится обычная каталожная запись с коротким именем, а длинное имя сохраняется в каталоге в записях, предшествующей обычной. Количество дополнительных строк каталога определяется длиной «длинного» имени файла.

Файлы и каталоги с длинным именем (свыше 8.3) обрабатываются файловой системой FAT особым образом. Структура 32-байтовой записи для файла с LFN (Long File Name) отличается от обычной (SFN-записи) — см. рис. 51.

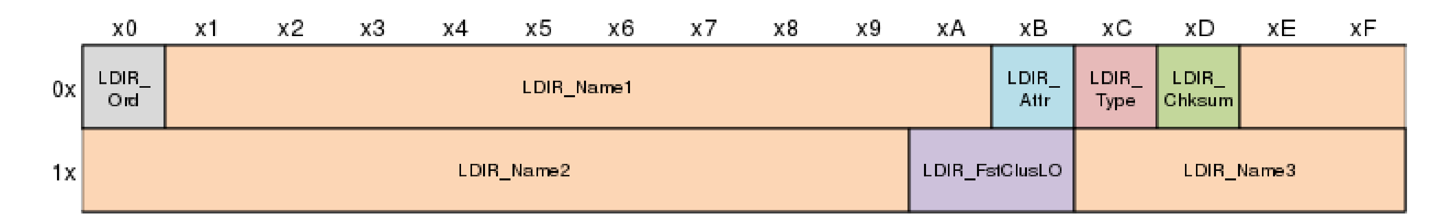


Рис. 51. Строка каталога файловой системы FAT-32 — LFN-запись.

Файловая запись FAT (LFN-запись) состоит из следующих полей:

- LDIR_Ord. Первый байт записи служит для нумерации записей в наборе.
- LDIR_Name1. 10-байтовое поле по адресу 0x01 содержит первые пять символов имени файла (вернее, той части его имени, которая отражена в данной LFN-записи).
- LDIR_Attr. Байт атрибута по адресу 0x0B, равен 0x0F (ATTR_LONG_NAME).
- LDIR_Type. Байт по адресу 0x0C, обнулён и дополнительно свидетельствует, что данная запись таблицы FAT относится к файлу с длинным именем.
- LDIR_Chksum. Байт по адресу 0x0D, содержит контрольную сумму SFN псевдонима файла, соответствующего набору LFN-записей.
- LDIR_Name2. 12-байтовое поле по адресу 0x0E, содержащее с 6-го по 11-й символы имени файла.
- LDIR_FstClusLO. 2-байтовое поле по адресу 0x1A, в контексте LFN-записи лишено смысла и обнуляется.
- LDIR_Name3. 4-байтовое поле по адресу 0x1C, содержащее 12-й и 13-й символы имени файла.

Набор LFN-записей каталога FAT всегда должен быть связан с обычной SFN-записью, которой физически предшествует на диске, то есть, LFN-запись идёт первой, а SFN-запись последней. Набор LFN-записей, обнаруженный без соответствующей обычной записи, называется орфаном, и запись считается повреждённой; подобный файл совершенно невидим в старых версиях MS-DOS/Windows.

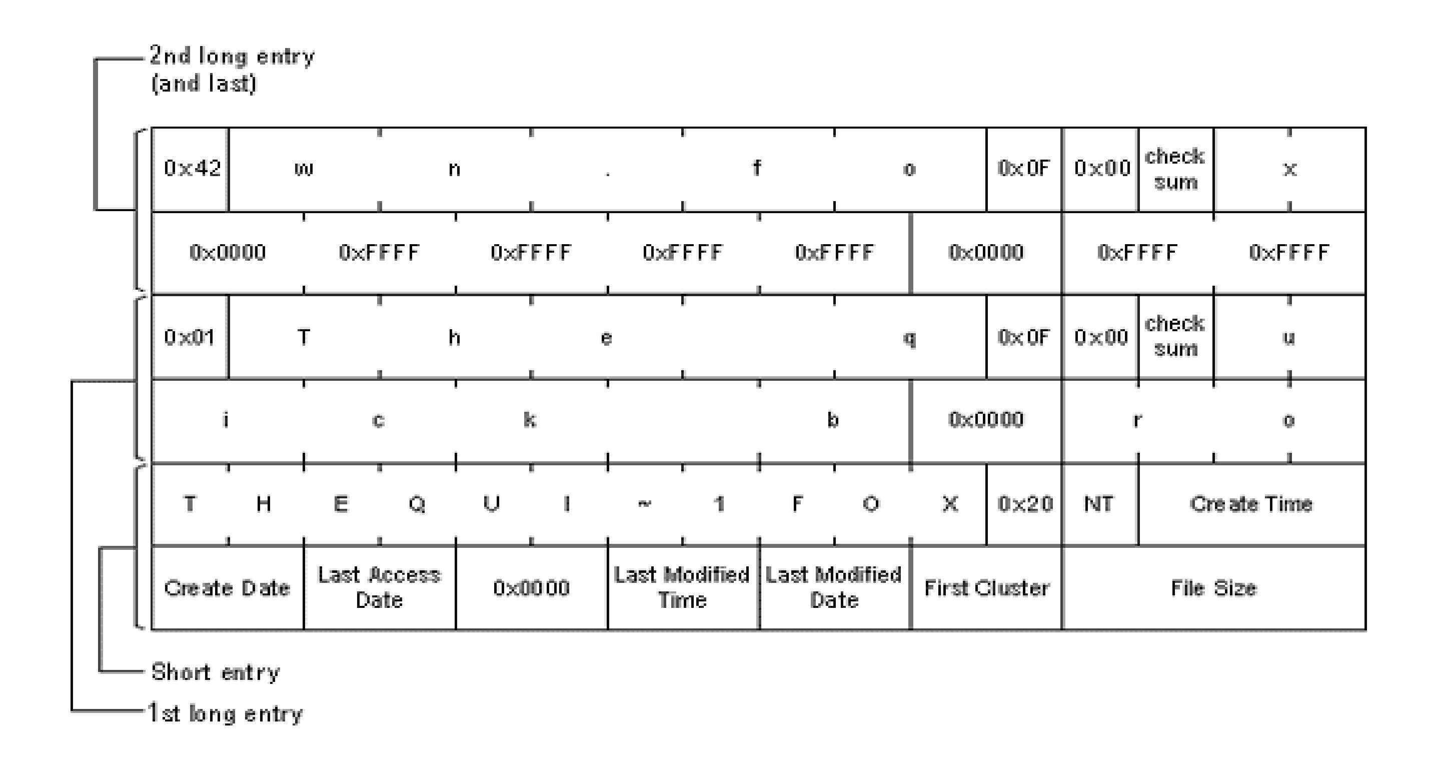


Рис. 52. Пример длинного имени (из документации на FAT от MS).

В последовательности LFN-записей каждая из них имеет собственный порядковый номер, определяемый первым байтом (LDIR_Ord). То есть, последней идёт запись SFN (см. рис. 52). Перед ней, со значением LDIR_Ord = 0x01, первая LFN-запись. Выше неё, вторая LFN-запись с LDIR_Ord = 0x42 (!). В данном примере используются только две LFN-записи (такое имя файла = «The quick brown.fox»). Если бы имя было подлиннее, то выше записи с кодом 0x42, была бы запись с кодом 0x43. То есть, для всех вышестоящих для выделения порядкового номера записи используется маска 0x40. Но! Непроверено. Возможно, это ошибка в документации.

Таким образом, длинное имя записывается в каталог первым, причем фрагменты размещены в обратном порядке, начиная с последнего. Вслед за длинным (полным) именем размещается SFN-запись файла, содержащая укороченный по специальному алгоритму вариант этого имени.

Длинные имена файлов хранятся в кодировке Юникод (UTF-16), при этом сохраняется вводимый регистр буквенных символов. Если некоторый символ имени в кодировке OEM или Юникоде не может быть превращён в символ кодовой страницы, он всегда отображается как символ подчёркивания «_», причём сохранённый на диске действительный символ не изменяется.

Байт контрольной суммы (LDIR_Chksum) вычисляется по определённому алгоритму на основе 8.3-имени обычной записи и копируется во все соответствующие ей «длинные» записи.

6.1.5. Алгоритмы файловых операций в FAT

Форматирование тома — таблица FAT обнуляется, за исключением первых трёх строк (FAT[0] и FAT[1], зарезервированы, а FAT[2] содержит запись, соответствующую файлу метки тома, либо, при отсутствии её — метку EOC) и записей повреждённых кластеров; записи корневого каталога обнуляются (за исключением файла метки тома, если она есть), в остальном область данных не затрагивается. Если раздел новый (не было на нём ещё файловой системы FAT), то структуры данных создаются вновь.

Удаление файла — первый символ файловой записи и всех ассоциированных LFN-записей заменяется кодом 0xE5; занимаемые файлом кластеры помечаются в таблице FAT как свободные, а кластеры в области данных не затрагиваются. То есть, кластеры с данными не обнуляются и все поля каталога, кроме 1-ого байта имени, сохраняются, файл легко восстановить.

Создание файла или каталога командой «Создать» контекстного меню — создаётся файловая запись для нового «пустого» файла с именем по умолчанию (например «Новая папка») и размером, определяемым типом файла; сам файл, если имеет ненулевой размер (что верно для практически всех «пустых» файлов, кроме каталогов и текстовых документов) записывается в области данных в выделенные ему кластеры; в таблице FAT создаётся соответствующая кластерная цепочка. После присвоения файлу действительного имени (не по умолчанию) первоначально созданная файловая запись помечается как удалённая и создаётся новая.

Переименование файла — создаётся новая запись с обновлённым именем; старая запись помечается как удаленная.

Сохранение файла из приложения (не из командной строки) — создаётся запись, содержащая все поля, кроме размера и начального кластера файла; после завершения сохранения файла создаётся новая запись, содержащая все поля, а прежняя удаляется.

Копирование файла — в новом местоположении создаётся идентичная файловая запись (возможно, за исключением некоторых временных отметок, см. выше), файлу выделяется первый свободный кластер и содержимое файла копируется в новое место, причём происходит копирование текущего кластера, поиск следующего свободного и заполнение таблицы FAT.

Перемещение файла (между разными томами/разделами) — копирование с последующим удалением файла из исходного местоположения.

Перемещение файла (в пределах тома/раздела) — кластерная цепочка не затрагивается, файловая запись копируется без изменения в новый каталог, после чего удаляется из прежнего.

Поиск свободного кластера по таблице FAT для выделения новому файлу начинается в общем случае не с начала области данных (то есть с кластера 2), а с последнего выделенного какому-либо файлу кластера, номер которого сохраняется в структуре FSInfo. Другими словами, если файлу 1 был отведён кластер 30, а файлу 2 — кластер 31, после чего файл 1 был удалён, то при создании нового файла 3 он, скорее всего, будет физически размещён начиная с кластера 32. То есть, удалённые файлы начнут заниматься под новые, только после исчерпания свободных, помеченных нулём.

6.1.6. ExFAT

ExFAT (*Extended FAT* — «расширенная FAT»), иногда называется *FAT64* — проприетарная файловая система, предназначенная главным образом для флэш-накопителей. Впервые представлена фирмой Microsoft для встроенных устройств в Windows Embedded CE 6.0. Размер кластера по умолчанию для файловой системы exFAT составляет от 4 КБ до 128 КБ в зависимости от размера тома, максимальный допустимый по спецификации — 32 МБ. Можно сказать, что это FAT32 со снятыми ограничениями.

Основными преимуществами exFAT перед предыдущими версиями FAT являются:

- уменьшение количества перезаписей одного и того же сектора, что важно для флеш-накопителей, у которых ячейки памяти необратимо изнашиваются после определённого количества операций записи (это сильно смягчается выравниванием износа (англ. *wear leveling*), встроенным в современные USB-накопители и SD-карты).
- теоретический лимит на размер файла 2^{64} байт (16 эксабайт).
- максимальный размер кластера увеличен до 2^{25} байт (32 мегабайта).
- улучшение распределения свободного места за счёт введения бит-карты свободного места, что может уменьшать фрагментацию диска.
- введена поддержка списка прав доступа[3].
- поддержка транзакций (опциональная возможность, должна поддерживаться устройством).

Существует свободный драйвер exFAT в виде патча для ядра Linux, поддерживающий только чтение этой файловой системы. Также существует драйвер, работающий через FUSE, в том числе для ОС FreeBSD и OpenBSD (*sysutils/fuse-exfat*). Данный драйвер поддерживает как чтение, так и запись. Кроме того, в августе 2013 года Samsung опубликовала драйвер для ядра Linux под лицензией GPL.

6.2. Файловая система ntfs

6.2.1. Структура файловой системы ntfs.

Для ускорения доступа и уменьшения объёма вычислений, сектора раздела объединяются в кластеры — смежные последовательности секторов. Размер кластера определён в диапазоне от 512 байт (1 сектор) до 64 килобайт (128 секторов).

В начале раздела находится загрузочная область (см. рис. 53), в терминологии Windows называемая загрузочной записью раздела (Volume Boot Record). В ней, в самом начале, содержится блок параметров BIOS (BPB) размером 73 байта — информация о разделе (см. ниже описание этого блока параметров; это аналог суперблока, используемого в unix/linux; см. таблицу 11), а далее до конца загрузочной области идёт код загрузки Windows — вторичный загрузчик.

Загрузочная запись занимает обычно 8 КБ (16 первых секторов). Говорят (в Интернете), что загрузочная область дублируется в самом конце раздела. Непроверено.

Таблица 11. Расширенная BPB для NTFS (73 байта).

Смещение в секторе	Смещение в BPB	Длина	Описание
0x00B	0x00	25 байт	DOS 3.31 BPB
0x024	0x19	1 байт	Номер физического диска
0x025	0x1A	1 байт	Флаги (идентичные DOS 3.4 EBPB)
0x026	0x1B	1 байт	Расширенная загрузочная запись (по аналогии с DOS 3.4 EBPB и NTFS EBPB)
0x027	0x1C	1 байт	Зарезервировано
0x028	0x1D	8 байт (qword)	Секторов в томе
0x030	0x25	8 байт (qword)	Первый кластер MFT
0x038	0x2D	8 байт (qword)	Первый кластер MFT (копия предыдущего поля)
0x040	0x35	4 байта (dword)	Размер записи MFT
0x044	0x39	4 байта (dword)	Размер индексного блока
0x048	0x3D	8 байт (qword)	Серийный номер тома
0x050	0x45	4 байта (dword)	Контрольная сумма

В определенной области раздела (адрес начала этой области указывается в загрузочной записи, точнее в BPB — см. таблицу 11) расположена основная системная структура NTFS – главная таблица файлов (Master File Table, MFT). В записях этой таблицы содержится вся информация о расположении файлов на разделе, а данные небольших файлов хранятся прямо в записях MFT — резидентно.

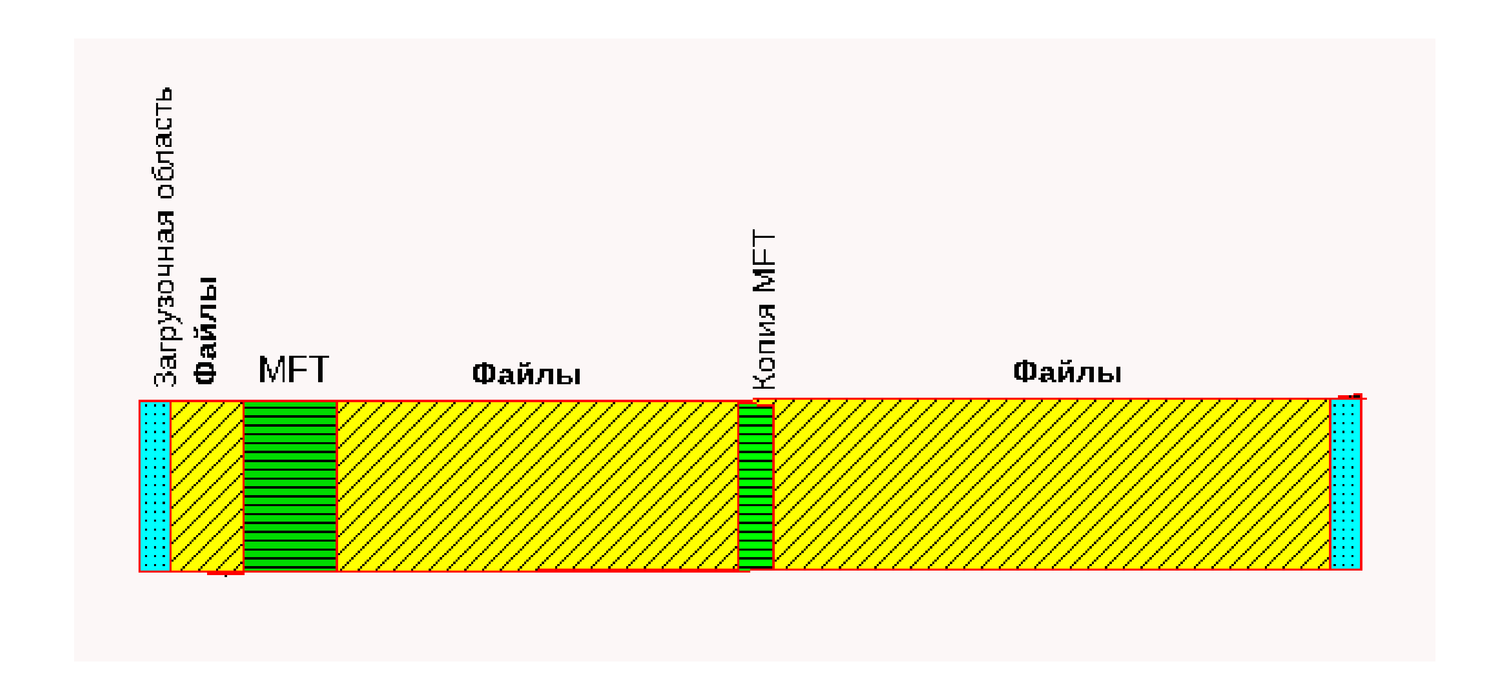


Рис. 53. Структура NTFS.

Начало MFT — первый 16 строк, для надёжности дублируются — их копия (файл $MFTMirr) располагается обычно в середине раздела. Точнее, по адресу, указанному во второй строке MFT — это не обязательно середина раздела.

Важной особенностью NTFS является то, что вся информация, как пользовательская, так и системная, хранится в виде файлов. Имена системных файлов начинаются со знака "$". Например, загрузочная запись тома содержится в файле $Boot, а главная таблица файлов – в файле $Mft. Такая организация информации позволяет единообразно работать как с пользовательскими, так и с системными данными на томе.

Поскольку MFT является важнейшей системной структурой, к которой при операциях с томом наиболее часто происходят обращения, выгодно хранить файл $Mft в непрерывной области логического диска, чтобы избежать его фрагментации (размещения в разных областях диска), и, следовательно, повысить скорость работы с ним. С этой целью при форматировании тома выделяется непрерывная область, называемая зоной MFT (MFT Zone) размером обычно 12% от объёма раздела. По мере увеличения главной таблицы файлов, файл $Mft расширяется, занимая зарезервированное место в зоне.

Остальное место (88%) на разделе NTFS отводится под файлы – системные и пользовательские.

Но! Если пользователи «засрут» своими файлами все предоставленное им 88% пространства раздела, то их файлы начнут создаваться и в зарезервированной MFT Zone. Это означает, что файл $Mft, как и обычные файлы, также может оказаться фрагментированным.

6.2.2. MFT

Первые шестнадцать элементов главной таблицы файлов MFT зарезервированы для специальных файлов. Пока используются только первые двенадцать элементов. Это скрытые файлы, имена которых расположены в корне раздела. Файлов не видно, но, тем не менее, они таки существуют. Проверить это можно, попытавшись создать файл с одним из зарезервированных имён в корне раздела.

Список специальных файлов NTFS

0 - $MFT (элемент 0, или строка 0 таблицы) - Главная таблица файлов. Атрибут данных содержит элементы MFT, а также неиспользуемые растровые атрибуты.

1 - $MFTMirr (элемент 1, или строка 1 таблицы) - Зеркало (резервная копия) первых шестнадцати элементов MFT.

2 - $LogFile (элемент 2) - Файл журнала, в который записываются все изменения при работе.

3 $Volume (элемент 3) - Атрибут данных $Volume представляет весь раздел. Обращение Win32 по имени «\\.\C:» откроет файл тома на диске C: (предполагается, что диск C: является разделом NTFS), Файл $Volume содержит также имя раздела, информацию о разделе и атрибуты идентификатора объекта.

4 - $AttrDef (элемент 4) - Атрибут данных $AttrDef содержит массив определений атрибутов, которын будут использоваться при описании файлов:

```
Typedef struct {
    WCHAR AttributeName[64];
    ULONG AttributeNumber;
    ULONG Unknown[2];
    ULONG Flags;
    ULONGLONG MinimumSize;
    ULONGLONG MaximumSize;
} ATTRIBUTE_DEFINITION, *PATTRIBUTE_DEFINITION;
```

5 - \ (элемент 5) - Корневой каталог файловой системы. Обычно он располагается в пользовательском пространстве раздела, то есть, за пределами 12-процентной зоны.
6 - $Bitmap (элемент 6) - Атрибут данных $Bitmap представляет собой битовую карту кластеров раздела.
7 - $Boot (элемент 7) - Первый сектор $Boot (вторичного загрузчика) является также и первым сектором раздела. Поскольку он используется в самом начале процесса загрузки системы (если раздел является загружаемым), то пространство здесь не нормируется, а хранимые данные не выравниваются по естественным границам. Формат первого сектора можно описать с помощью структуры BOOT_BLOCK:

```
#pragma pack(push, 1)
typedef struct {
    UCHAR Jump[3];
    UCHAR Format[8];
    USHORT BytesPerSector;
    UCHAR SectorsPerCluster;
    USHORT BootSectors;
    UCHAR Mbz1;
    USHORT Mbz2;
    USHORT Reserved1;
    UCHAR MediaType;
    USHORT Mbz3;
    USHORT SectorsPerTrack;
    USHORT NumberOfHeads;
    ULONG PartitionOffset;
    ULONG Reserved2[2];
    ULONGLONG TotalSectors;
    ULONGLONG MftStartLcn;
    ULONGLONG Mft2StartLcn;
    ULONG ClustersPerFileRecord;
    ULONG ClustersPerIndexBlock;
    ULONGLONG VolumeSerialNumber;
    UCHAR Code[0x1AE];
    USHORT BootSignature;
} BOOT_BLOCK, *PBOOT_BLOCK;
#pragma pack(pop)
```

8 - $BadClus (элемент 8) - В атрибуте данных этого файла содержится информация о сбойных кластерах.
9 - $Secure (элемент 9) - Атрибут данных $Secure содержит совместно используемые идентификаторы доступа. $Secure содержит также два индекса.
10 - $UpCase (элемент 10) - Атрибут данных $Upcase содержит эквивалент верхнего регистра всех 65536 символов Unicode. Это нужно для хранения имён файлов по стандарту POSIX — с различием маленьких и больших букв.
11 - $Extend (элемент 11) - $Extend — это каталог, который содержит специальные файлы, используемые некоторыми дополнительными функциями NTFS. Специальные файлы, хранящиеся в этом каталоге, это: «$ObjId» (поддержка объектных идентификаторов), «$Quota» (поддержка квот), «$Reparse» (данные точек повторной обработки) и «$UsnJrnl» (журнал файловой системы). Начиная с Windows Vista здесь также находится каталог «$RmMetadata» (поддержка транзакций NTFS).

Остальные 4 строки MFT — резерв.

Эти 16 строк — аналог суперблока и в силу их важности для живучести файловой системы, они имеют дубль - файл $MFTMirr в середине раздела. По слухам из источников, обычно поставляющих достоверную информацию, на самом деле файл $MFTMirr содержит не все 16 первых записей MFT, а только первые четыре. Тем не менее, это необходимо проверить.

Далее в MFT идут ещё несколько пустых строк, до 23-ой (считаем с нуля). Эти строки создаются при форматировании раздела.

А, вот, дальше, начиная с 24-ой, строки в MFT создаются динамически при создании файлов пользователем.

Пример строки MFT, определяющей некоторый не очень большой файл, приведён на рис. 54. В атрибуте «имя_файла» хранится имя файла, несколько атрибутов и ссылка на каталог, в котором файл зафиксирован.

В NTFS есть существенные отличия в хранении малых файлов, средних, больших, сверхбольших и огромных. Также на хранение файла влияет фрагментированность файла. Например, если файл настолько велик (или сильно фрагментирован, или с файлом часто работали и у него большая история изменений), что его атрибуты не помещаются в одной строке MFT, то эти атрибуты хранятся нерезидентно: из текущей строки MFT заголовок атрибута ссылается на некоторую другую строку MFT, возможно не рядом расположенную, где хранятся данные атрибута. Дополнительных строк у описания файла может быть до 4-х штук. Если и их не хватает, тогда для хранения атрибутов файла привлекаются кластеры в разделе.

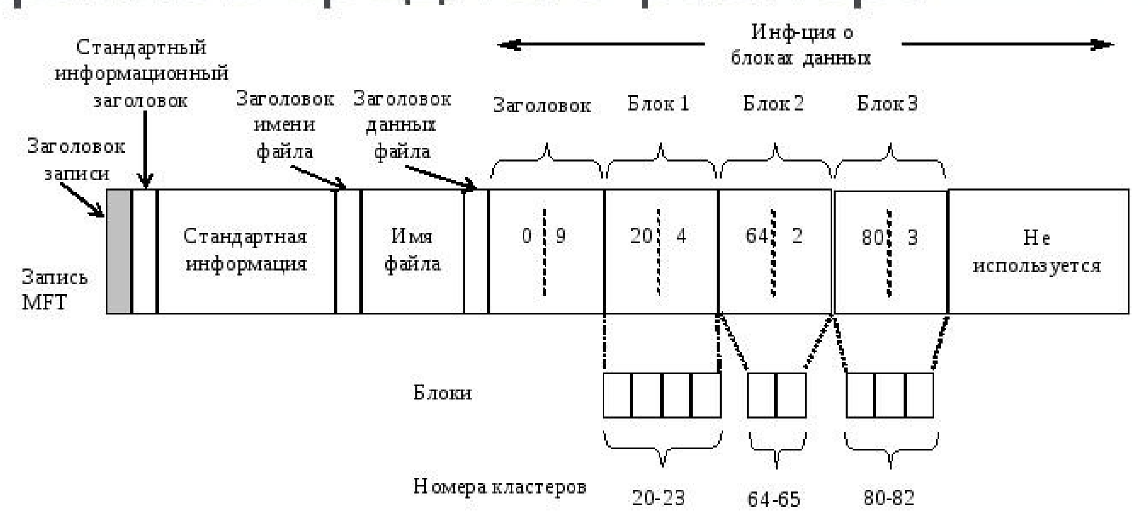


Рис. 54. Строка MFT, определяющая файл среднего размера — от нескольких килобайт.

6.2.3. Каталоги

Каталог в файловой системе ntfs представляется записью MFT (см. рис. 55), у которой установлен специальный флаг в заголовке записи и в атрибутах $STANDART_INFORMATION и $FILE_NAME, а по содержанию — это таблица, которая содержит следующие графы:

- индекс файла — номер строки MFT, в которой содержатся атрибуты файла,
- длина имени файла — длину «длинного виндового» имени (уточняем, потому что в файловой системе ntfs файл имеет три имени — имя msdos в формате 8.3, имя POSIX - unix-овое с разичием регистра символов и имя «виндовое» - в котором большие и маленькие буквы не различаются), она может от 1 до 255,
- само длинное «виндовое» имя файла,
- поле основных атрибутов файла, в число их входят: временные штампы, размеры потоков файла и некоторые флаги.

Однако хранится «таблица» каталога как списковая структура, как последовательность записей переменной длины — ведь, имена файлов разной длины. Для того, чтобы найти конец одной записи каталога и начало другой, используется поле «длина имени файла».

Если каталог маленький (не более нескольких файлов, сколько — определяется версией ntfs), то он хранится «резидентно», то есть, в самой записи MFT, описывающей файл каталога — хотя в этом случае файла каталога как раз может и не быть, всё — в записи (см. рис. 55). Но поскольку в ntfs «всё есть файл», то в предыдущем предложении противоречия нет.

Но даже если каталог хранится резидентно, некоторые атрибуты его вполне могут храниться в кластерах где-то в другом месте.

Если каталог большой, то начальная часть его хранится резидентно в MFT, а остальная часть является обычным файлом, помеченным специальным атрибутом в строке MFT. Кроме того, большие каталоги хранятся в кластерах раздела в формате бинарного дерева. Пример каталога среднего размера (на 9 файлов) показан на рисунке 56. Красные стрелки показывают, что каталожные записи хранятся в алфавитном порядке. Корень B+ дерева находится в атрибуте IndexRoot. Поскольку в строке MFT резидентно девять каталожных записей не помещаются, то для их хранения выделяются два кластера с виртуальными номерами VCN0 и VCN2. Эти кластеры расположены в разделе по адресам, соответственно, 200 и 40.

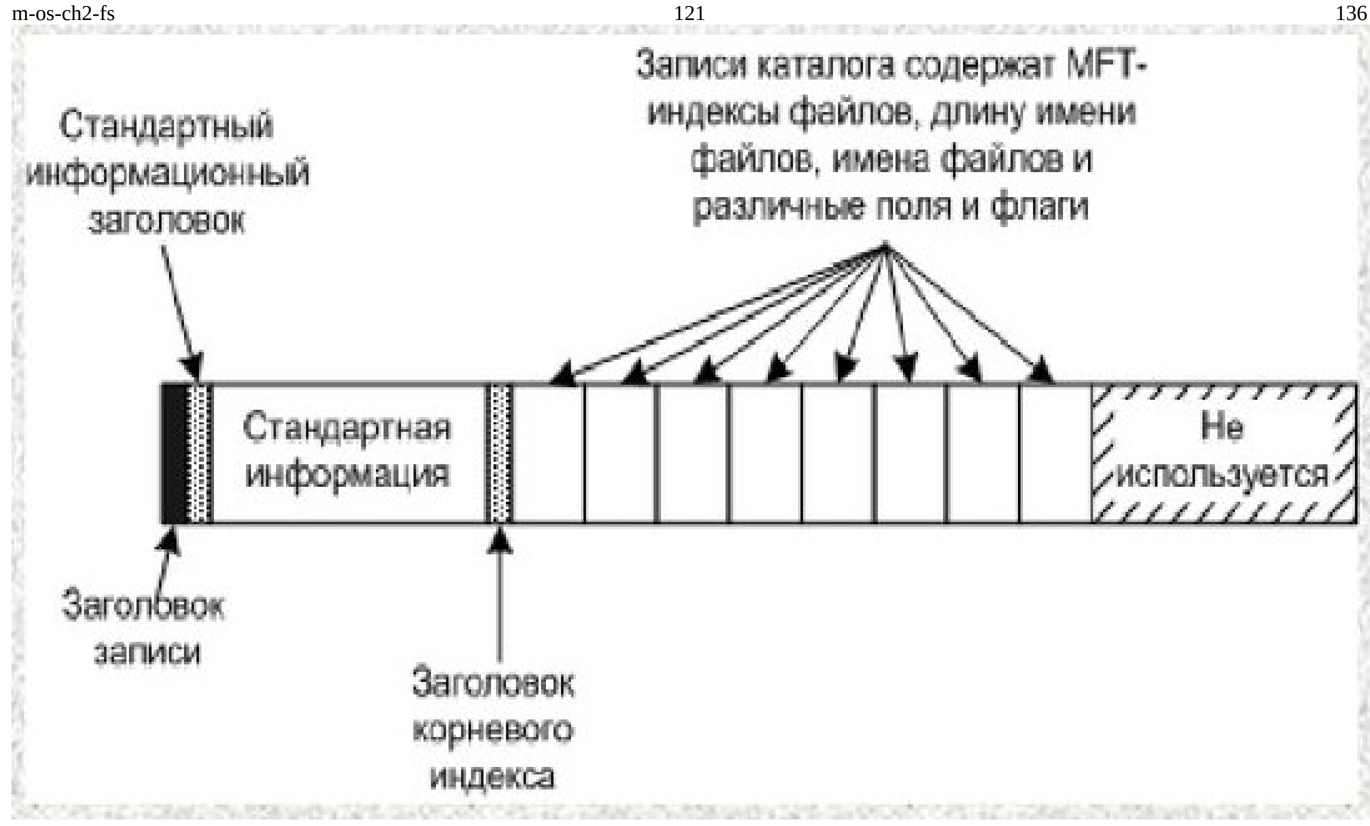


Рис. 55. Строка MFT, содержащая запись о каталоге.

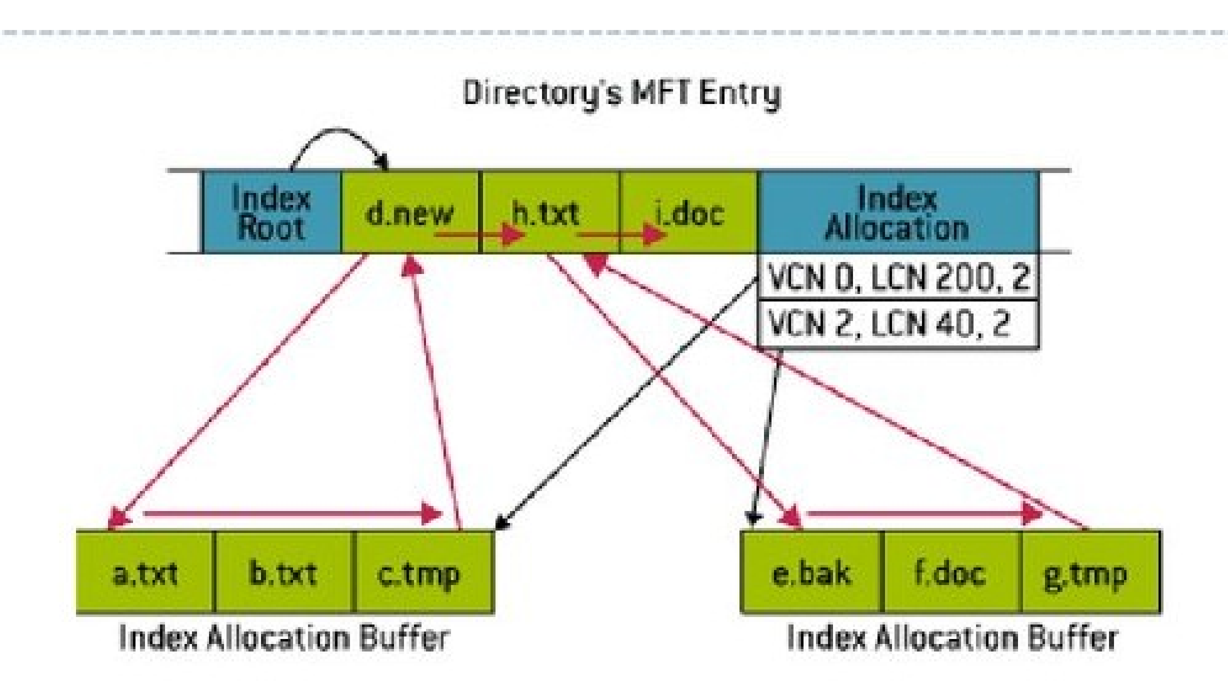


Рис. 56. Хранение в MFT каталога среднего размера.

6.2.4. Некоторые возможности ntfs

Масштабируемость. Как свидетельствует опыт использования ОС Windows на суперкомпьютерах проблемы с масштабируемостью есть и намалые. В настоящее время в Top-500 нет ни одного суперкомпьютера с Windows.

Журнал файловой системы. Имеет место быть несовместимость журналов в разных версиях системы, а реестре даже существуют настройки, предотвращающие обновление журнала до новых версий.

Жёсткие ссылки. В ntfs они определены и даже иногда используются, например, для поддержки коротких (8.3) имен файлов. Файловая система ntfs имеет ограничение в 1024 жесткие ссылки на файл. Пользователями эта функциональность не используется.

Мягкие ссылки (soft). Также есть и используются достаточно широко, в том числе пользователями.

Альтернативные потоки данных. Они позволяют использовать более одного потока данных быть связанным с именем файла в формате «имя файла: имя потока» (например, «текст.txt: дополнительный поток»). Альтернативные потоки не отображаются в проводнике Windows, и их размер не входит в размер файла. Когда файл копируется или перемещается в другую файловую систему без поддержки потоков, пользователь получает предупреждение о невозможности сохранения альтернативных потоков данных. Такое предупреждение обычно не появляется, если файл прикреплен к электронному письму или загружен на веб-сайт. Таким образом, использование альтернативных потоков для критических данных может вызвать проблемы. Microsoft предоставляет инструмент под названием Streams для просмотра потоков на выбранном томе. Также это умеют делать плагины для файлового менеджера Total Commander, Крайне опасная вещь.

Сжатие файлов. NTFS может сжимать файлы по алгоритму LZNT1 (вариант LZ77), но только при использовании кластеров небольшого размера, а сжатие больших файлов к тому же неэффективно.

С Windows 10 Microsoft представила дополнительные алгоритмы сжатия, которые используются, например, для сжатия всего системного раздела, что актуально, например, для планшетов и смартфонов.

Поддержка разреженных файлов. Имеется функциональность для эффективного хранения таких файлов, что позволяет экономить место.

Шифрование разделов и файлов. Есть.

Квоты на использование дискового пространства для пользователей. Есть.

6.2.5. Некоторые недостатки ntfs

а) Авторы полностью согласны с утверждением, файловая система NTFS существенно более надёжна, чем . . . чем что?
Правильно! Чем FAT.

И не более того.

До надёжности файловых систем EXT4, JFS, XFS, reiserfs и тем более UFS — ей как до Альфы Центавра. Это не шутка. Если вы хоть что-то соображаете в теории надёжности, то посмотрите на структуры этих файловых систем и ответьте:

- как может считаться надёжной система, если она свои структуры данных хранит в одном экземпляре?

- во сколько раз различается надёжность хранения данных в одном экземпляре от надёжности распределённого хранения данных?

Журнал, говорите? Так ведь, в EXT4, JFS, XFS, reiserfs и в последних версиях UFS также есть журналы. Причём, их журналы способны работать в реэиме journal, про который система журналирования NTFS просто не знает, она работает только в режиме ordered.

б) Запутанность. Это исключительно запутанная файловая система. Складывается впечатление, что её однажды, в самом начале 90-х, сделали специалисты, а потом пришли молодые «временщики» и, не вникая и не разбираясь во всяких там концепциях, принципах, положениях, начали «сопровождать» методом «залепливания дыр», получив в результате «блюдо спагетти», в котором сам чёрт ногу сломит и без (да-да, этого!) не разобраться. Косвенным доказательством этого служит начало разработки файловой системы «нового поколения» ReFS, обратно совместимой с NTFS: похоже «клиент созрел» до состояния, когда сопровождать становится просто тяжко.

Думаете, авторы запугивают? Ну так попробуйте разобраться в структуре записей MFT о файлах или каталогах. Интернет вам в руки. Крис Касперски разобрался. Его публикации в Интернете доступны. Попробуйте повторить. Как сможете, придёте, поговорим.

в) Пункт б) приводит к тому, что накладных расходов при функционировании файловой системы стало слишком много: и процессорного времени, и памяти. Сама MS не рекомендует использовать NTFS на малых разделах, на которых служебные данные (прежде всего MFT) могут занимать до 25% объёма раздела.

Действительно, сравним:

- ext4, UFS2 — строка индексной таблицы 256 байт и она не полностью занята, есть место для резидентных файлов,

- NTFS — строка MFT от одного до четырёх килобайт, описание файла может занимать до 4-х строк (до 4-х килобайт!) и причём, очень часто, в строке MFT находится только заголовок атрибута, а сами значения — где-то там в кластерах на разделе. То есть запись о файле может многократно превышать размер строки MFT.

То есть, в NTFS объём служебной информации о файле в 4-16 и более раз больше, нежели в ext4, UFS2. Причём, как правило, информация о файлах хранится в форме «имя=значение», то есть, имя текстом и значение иногда текстом, информация обрабатывается как текстовые строки, что в десятки раз более трудоёмко, нежели обработка чисел. Не знают в MS такого понятия, как «классификатор».

г) Слабая защита и потенциальная «дырявость», прежде всего, из-за реализованной многопоточности файлов — сколько «баз данных» утекло из-за этого в Интернет, целый бизнес на этом построен.

д) Фрагментация, что особенно прискорбно — каталогов. С большой вероятностью средние и, тем более, большие каталоги будут фрагментированы. Просто потому, что каталоги меняются в размерах гораздо чаще, чем файлы. Более того, учитывая, что атрибуты файлов из MFT и каталогов разбросаны по кластерам — это почти всегда так. Следствие — замедление работы и необходимость постоянного отслеживание параметра «фрагментированность».

Пусть имеем множество всех возможных комбинаций имён файлов из маленьких латинских букв длиной две буквы (из двух по два). Это будет всего около 700 файлов. Пусть размер блока файловых систем **UFS2/ext4** равен 4 Кб. Тогда каталог с именами этих файлов займёт примерно 2 блока. Ничего «прилагательного» к каталогу в этом случае нет. И «лежать» всё это добро будет рядышком в одной группе_цилиндров/одной_группе+блоков. Кстати и сами файлы будут тут же рядом.

Пусть то же самое имеем в **ntfs**. Гложат меня смутные сомнения, что каталог со всем прилагательным (оно тут есть, просто потому, что каталог в ntfs сложнее устроен — см. выше) займёт с десяток кластеров. И этот десяток будет разбросан по всему (!фанфары) разделу ntfs. И файлы, кстати, тоже будут разбросаны.

Особенно будет всё всё выглядеть мрачно, если с файлами активно работать: изменять, переименовывать, копировать, перемещать, удалять и создавать заново и т. д.

Для садомазохистов самое то. . . Поэтому MS озадачилась созданием новой файловой системы — ReFS.

А ntfs, заполненная на 80% - 90% - это вообще катастрофа.

Вопросы на «засыпку»

1. Если у вас в компьютере установлены один SSD с одним разделом и один винчестер с двумя разделами, то сколько у вас корневых каталогов?

2. В файловой системе ntfs «всё есть файл». Однако, из-за специфики выделения места под файл при его создании, файлы часто оказываются фрагментированными. MFT — тоже файл, но под неё при форматировании резервируется 12% дискового пространтства. Может ли MFT оказаться фрагментированной?

3. Какая разница между понятиями «диск», «раздел», «том».

4. Что означает число 32 в имени файловой системы FAT32?

5. Как с помощью Total Commander увидеть дополнительные потоки файла?

6.

ЛЕКЦИЯ 7. ДРУГИЕ ФАЙЛОВЫЕ СИСТЕМЫ

7.1. Файловая система CD-дисков (iso9660)

1. ISO 9660 — стандарт, выпущенный Международной организацией по стандартизации, описывающий файловую систему для дисков CD-ROM. Также известен как CDFS (Compact Disc File System).

Файловая система CDFS появилась почти при царе Горохе в незапамятные времена в конце 70-х годов вместе с изобретением CD-дисков или чуть позже. Первоначально CD-диски нашли широкое применение для замены грампластинок («виниловых»), то есть, для хранения аудиозаписей. А поскольку на грампластинках сохранялось 10-15 записей на двух сторонах, то аналогичные требования были сформулированы и для CD-дисков — хранить 10-20 записей.

Это применение CD-дисков было исключительно популярным в 80-е годы и в 1988 году был согласован и принят международный стандарт ISO 9660, давший файловой системе CDFS международный статус.

С другой стороны, в 80-е годы наблюдалось широкое шествие в народных массах операционной системы msdos с файловой системой FAT. Эти два процесса между собой тесно взаимодействовали, что привело к тому, что файловая система CDFS была создана существенно совместимой с файловой системой FAT и одновременно в ней появились те же минусы, что наличествовали в FAT. В частности: имена файлов ограничены восемью символами и тремя символами расширения; в именах используются только буквы латинского алфавита; фрагментация файлов не допускается, файл может располагаться только в непрерывной цепочке секторов; имена каталогов должны содержать не более 8 символов; максимальная глубина вложенных каталогов — до 8; максимальный размер файла — 2 Гб и др. ограничения.

Поэтому в 90-е годы с появлением и распространением в ширнармассах на ПЭВМ операционных систем Windows, MAC OS, linux и различных версий unix стандарт был доработан так, чтобы обеспечить совместимость с файловыми системами этих операционных систем — появился стандарт ISO 9660:1999, в котором для обеспечения совместимости со сложными файловыми системами были использованы «расширения» Romeo, Joliet, HPS, Rock Ridge, El Torito, что обеспечивало совместимость со всеми распространёнными файловыми системами. В частности:

- сняты ограничения на длинные имена файлов (разрешено до 255 символов);
- меньше ограничений на использование символов в именах файлов;
- структура каталогов произвольной вложенности.;
- для каждого файла записываются атрибуты:
    - код типа файла,
    - права доступа к файлу, в том числе поля uid и gid,
    - количество жёстких ссылок на файл,
    - времена создания, модификации, доступа, изменения атрибутов и др;
- поддерживаются специальные файлы:
    - разрежённые файлы,
    - символьные ссылки,
    - файлы устройств,
    - файлы сокетов,
    - FIFO-файлы;

- код приложения, в котором был создан файл, т. н. Creator code, определяющий, в каком приложении будет открыт данный файл;
- флаги и информация для отображения в файловом менеджере Finder;
- и др.

В настоящее время программы записи/создания CD/DVD, как правило, включают по умолчанию все расширения, а расширение El Torito используется в случае создания live-CD/DVD.

2. Общая структура *файловой системы ISO 9660 по стандарту 1988 года*

Диск может быть разбит на логические разделы - «сессии», но дальше рассматривается диск с одним разделом.
Запись осуществляется последовательно, по спирали; сектора по 2048 байт.
Порядок записи информации:
1. Каждый CD-ROM начинается с 16 пустых блоков (неопределенных ISO 9660), эта область может быть использована для размещения загрузчика ОС или для других целей. Первоначально эта область появилась из-за опасения, что начало диска может оказаться сбойным, всё-таки технология была новой. Но потом выяснилось, что эта пустая область (целых 32 Кб) очень даже нужна: в эту область начали помещать программы автозапуска CD/DVD, загрузчики для live-CD и др.
2. Дальше один блок **основного описателя тома** - хранит общую информацию о CD-ROM, в нее входит:
- идентификатор системы (32байта)
- идентификатор тома (32байта)
- идентификатор издателя (128байт)
- идентификатор лица, подготовившего данные (128байт)
- имена трех файлов, которые могут содержать краткий обзор, авторские права и библиографическая информация.
- ключевые слова: размер логического блока (как правило 2048, но могут быть 4096, 8192 и т.д.); количество блоков; дата создания; дата окончания срока службы диска.
- описатель **корневого каталога** (номер блока содержащего каталог).
3. Могут быть дополнительные описатели тома, подобные основному.
4. Корневой каталог. Структура каталога показана на рис. 57.

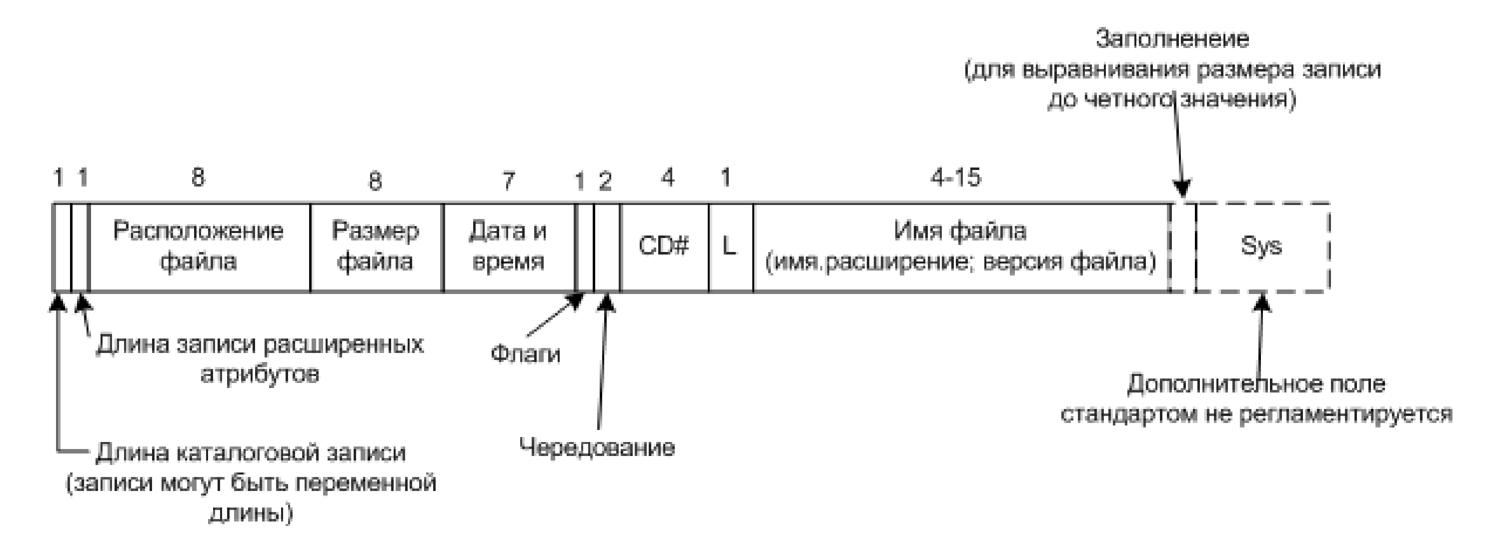


Рис. 57. Каталоговая запись стандарта ISO 9660.

Расположение файла - номер начального блока, так как блоки располагаются последовательно.
L - длина имени файла в байтах
Имя файла - 8 символов, 3 символа расширения (из-за совместимости с MS-DOS). Имя файла может встречаться несколько раз, но с разными номерами версий.
Sys - поле System use (используется различными ОС для своих расширений; именно сюда помещаются расширения Romeo, Joliet, HPS, Rock Ridge, El Torito).

Порядок каталоговых записей:
1. Описатель самого каталога (аналог ".")
2. Ссылка на родительский каталог (аналог "..")
3. Остальные записи (записи файлов) в алфавитном порядке
Количество каталоговых записей не ограничено, но ограничено количество вложенности каталогов - 8.

3. UDF — более современная файловая система, пришедшая на смены ISO-9660. Работает с CD, DVD и BD дисками., работает с файлами большого объёма, позволяет создавать мультисессионные диски.

4. Расширения

4.1. Joliet — расширение файловой системы ISO 9660. Спецификация была разработана фирмой Microsoft и поддерживается всеми версиями ОС Microsoft Windows со времён Windows 95 и Windows NT 4.0. Главной целью было ослабление ограничений на имя файла, накладываемых ISO 9660.

Joliet достигает этой цели введением дополнительного набора имён файлов (до 64 символов Unicode длиной), в кодировке UCS-2. Эти имена хранятся в специальном дополнительном заголовке (Supplementary Volume Descriptor, SVD), который безусловно игнорируется ISO 9660-совместимыми программами, обеспечивая этим обратную совместимость.

Большинство существующих программных платформ, включая Microsoft Windows, Linux, Mac OS X, и FreeBSD, способны читать носители информации с расширением файловой системы Joliet, что позволяет обмениваться файлами между этими операционными системами, даже при использовании нелатинских алфавитов (таких как Арабский, Японский, Кириллица), что было невозможно при помощи обычного ISO 9660.

4.2. Romeo расширения для Windows - стандарт Romeo предоставляет другую возможность записи файлов с длинными именами на компакт-диск. Длина имени может составлять 128 символов, однако использование кодировки Unicode не предусмотрено. Альтернативные имена в этом стандарте не создаются, поэтому программы MS-DOS не смогут прочитать файлы с такого диска.

Этот стандарт Romeo можно выбирать только в том случае, если диск предназначен для чтения приложениями Windows.

4.3. Rock Ridge Interchange Protocol (RRIP, IEEE P1282) — расширение файловой системы ISO 9660, разработанное для хранения файловых атрибутов, используемых в операционных системах POSIX (в том числе, unix/linux). Расширения Rock Ridge записываются поверх файловой системы ISO 9660 так, что оптический диск с Rock Ridge может быть прочитан программным обеспечением, рассчитанным на работу с ISO 9660 (см. поле Sys на рис. 57).

Rock Ridge может хранить следующую дополнительную информацию о содержимом диска:

- длинные имена файлов (до 255 символов);
- меньше ограничений на использование символов в именах файлов;
- структуру каталогов произвольной вложенности.
- для каждого файла записываются атрибуты:
- права доступа к файлу, в т. ч. поля uid и gid;
- количество жёстких ссылок на файл;
- времена создания, модификации, доступа, изменения атрибутов и др.
- поддерживаются специальные файлы:
- разрежённые файлы;
- символьные ссылки;
- файлы устройств;
- файлы сокетов;
- FIFO-файлы.

Эти данные записываются в специальные каталоги, имена которых обычно скрываются.

4.4. El Torito Bootable CD Specification — файловая система загрузочных дисков по стандарту ISO 9660. Формат был впервые представлен на публике в 1994 году, а впервые был использован уже в январе 1995 года.

BIOS сканирует все дисковые системы компьютера и, в соответствии со стандартом ISO 9660, загрузочный код диска эмулируется как жёсткий диск (код 80) или флоппи-диск (код 00),

после чего загрузка информации происходит в штатном режиме.

Термин El Torito, взятый из спецификации Phoenix/IBM Bootable CD-ROM Format Specification, в действительности является названием ресторана, расположенного рядом с офисом Phoenix Software (en:Phoenix Technologies). В ресторане „El Torito“ обычно обедали инженеры, занимавшиеся разработкой этого стандарта. Для пользователей ПК стандарт El Torito означал в первую очередь возможность загрузки с компакт-дисков и DVD, что открывало ряд новых возможностей, к которым относятся создание загрузочных «аварийных» дисков CD-ROM/DVD, загрузка с диска, содержащего новейшую версию операционной системы, при инсталляции последней в новых системах, создание загрузочных диагностических/тестовых компакт-дисков и многое другое.

4.5. Apple ISO9660 Extensions — алгоритм представления данных, записанных на диск в файловой системе HFS/HFS+, используемый в системах Macintosh, что позволяет пользователям данной ОС сохранять больше метаданных. При использовании данных дисков в других ОС, формат читается как стандартный ISO9660.

7.2. Файловая система CP/M

CP/M (Control Program for Microcomputers) - операционная система, предшественник MS-DOS, то есть, ещё более древняя, чем MS DOS. Лицензия на неё была куплена MS перед началом работы над MS DOS и послужила прообразом для последней.

В ее файловой системе только один каталог, с фиксированными записями по 32 байта.

Имена файлов - 8+3 символов верхнего регистра.

После каждой перезагрузки рассчитывается битовый массив занятых и свободных блоков. Массив находится постоянно в памяти (для 180 Кбайтного диска 23 байта массива). После завершения работы, он не записывается на диск.

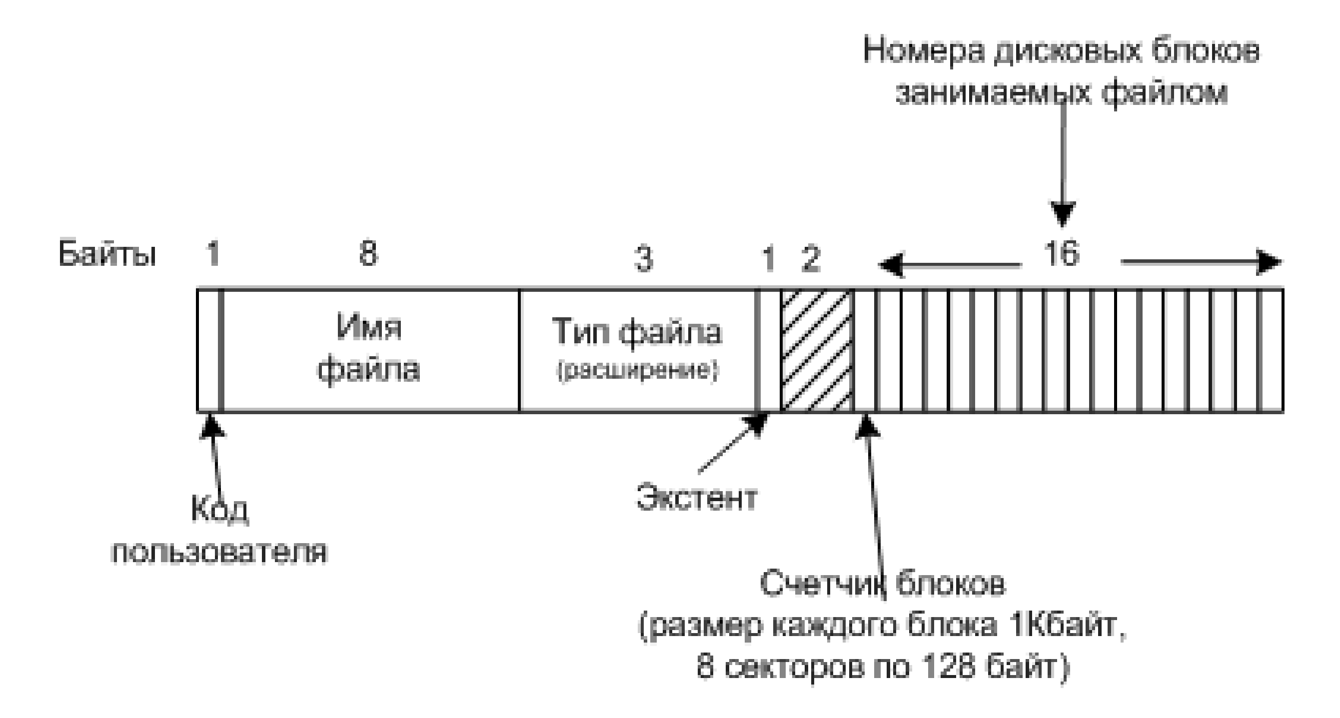


Рис. 58. Каталог CP/M.

Из рисунка 58 видно, что максимальный размер файла 16Кбайт (16*1Кбайт).

Для файлов размером от 16 до 32 Кбайт можно использовать две записи. Для файлов до 48

Кбайт три записи и т.д.

Порядковый номер записи хранится в поле **экстент**.

Код пользователя - каждый пользователь мог работать только со своими файлами.

Порядок чтения файлов:

- файл открывается системным вызовом open

- читается каталоговая запись, из которой получает информацию о всех блоках.

- вызывается системный вызов read, читаются блоки файла.

7.3. Файловая система смарт-карт

Смарт-карта: симка, банковская карта, кредитная карта (что выдают банки или крупные торговые структуры), карты доступа (пропуска) и т. д.

Минимальной логической единицей информации в смарт-карте в соответствии со стандартом ISO/IEC 7816 является *элемент данных* (DE - Data Element). В физической памяти элемент данных представлен *объектом данных* (DO - Data Object). Каждый объект данных представлен тремя полями:

- тегом (tag) - кодирует класс, тип и идентификатор объекта данных;
- длиной (length);
- значением (value).

Совокупность элементов данных образует файлы. Каждый файл имеет свой номер или идентификатор, состоящий из четырех 16-разрядных цифр.

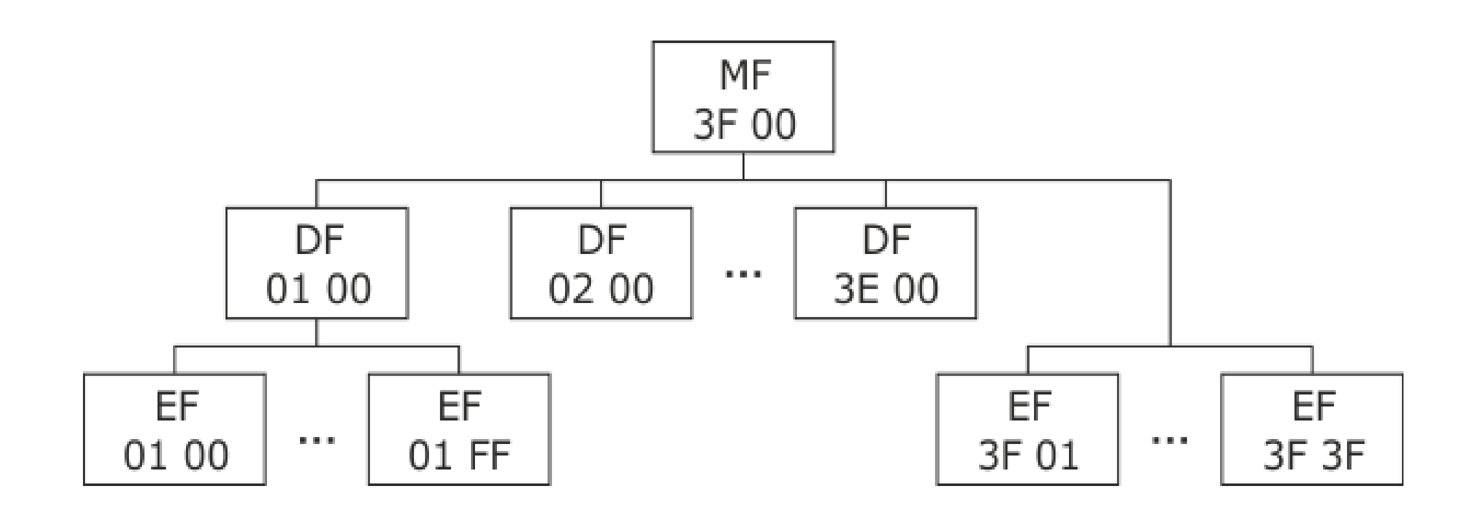


Рис. 59. Структура файловой системы смарт-карты.

В смарт-картах существуют два типа файлов, приведенных на рис. 59.

1) DF (Dedicated File) - файлы-директории. Определяют разделы пользовательской части EEPROM и содержащие другие файлы. Признаком того, что файл является директорией, являются нулевые значения двух последних цифр номера. Согласно стандарту ISO/IEC

7816, файловая система может поддерживать до 62 DF-файлов (01 00, 02 00, …, 3E 00). Смарт-карта должна содержать как минимум один файл DF, который называется MF - Master File. Данный файл имеет номер 3F 00 и является корнем древовидной структуры, которая олицетворяет собой файловую систему.
2) EF (Elementary File) - элементарный файл. Содержит данные смарт-карты. Каждый элементарный файл должен принадлежать DF-файлу. Принадлежность к определенному DF-файлу отображается в первых двух цифрах номера элементарного файла, которые повторяют первые цифры номера DF-файла, "потомком" которого является данный EF-файл. Каждый DF-файл (включая MF) может содержать до 63 EF-файлов.

Все файлы EF, содержащие полезные данные для конкретного приложения, всегда группируются вместе в одном каталоге DF. Это прием создает понятную структуру и облегчает ввод нового приложения в смарт-карту путем создания соответствующего DF. В смарт-карте с одним приложением (например, в простых телефонах) все файлы EF могут быть помещены либо прямо под главным файлом MF либо в отдельном каталоге DF. Смарт-карты с несколькими приложениями (например, в смартфонах) имеют соответствующее число каталогов DF, в которые помещаются файлы EF, принадлежащие конкретным приложениям.

Информация о файле, будь то DF-, MF- или EF-файл, хранится в заголовке файла, который носит название FCI - File Control Information, и представлена в таблице 12.

Таблица 12. Содержимое заголовка файла для DF- и EF-файлов.

Тег	**Содержимое объекта данных**	
	DF-файл	**EF-файл**
81h		File Size - размер файла
82h	File Descriptor - дескриптор файла	File Descriptor - дескриптор файла
83h	File Identifier - идентификатор файла	File Identifier - идентификатор файла
84h	DF Name - имя файла	DF Name - имя файла
85h	DF Attributes - атрибуты файла	DF Attributes - атрибуты файла
86h	Условия создания файлов-потомков	Условия доступа

Существуют три типа элементарных файлов:

- *Secret* - секретные файлы, предназначены для хранения ключевой информации;
- *Working* - рабочие файлы, используются для хранения данных, необходимых для организации взаимодействия с внешними приложениями (терминалом и хостом эмитента);
- *Internal* — внутренние файлы, предназначены для хранения данных, необходимых для работы приложений карты (счетчики транзакций, кошельки, сертификаты).

То есть, файловая система является иерархической, существует корневая папка (Master File, MF), обычные папки (Dedicated File, DF) и, собственно, файлы (Elementary File, EF). Каждый объект файловой системы имеет идентификатор (File ID, FID), который, в простейшем случае, состоит из четырех шестнадцатиричных цифр и который заменяет ему имя. В частности, всегда имеется идентификатор 3F00, принадлежащий корневому каталогу. В рамках каждого карточного стандарта существует свой перечень зарезервированных файловых идентификаторов, например, файл адресной книги на SIM имеет идентификатор 6F3A, а файл для хранения SMS — 6F3C. У стандартных файлов также имеются и символьные имена (исключительно для удобства человека, API их не использует и в файловой системе их нет; их создаёт и обрабатывает соответствующее приложение).
В случае с SIM-картами стандартные сценарии использования не предусматривают модификации файловой структуры, т.е. невозможно создать или удалить файлы.
Возможно только модифицировать содержимое файлов. Конечно, жизнь диктует и нестандартные сценарии, то есть, бывает.

Типы файлов. API смарт-карт предусматривает манипуляции с тремя типами файлов:

1) Transparent - аналог обычного бинарного файла любой файловой системы.
Средства работы с такими файлами — пара команд READ BINARY/UPDATE BINARY.

2) Linear Fixed **- ф**айл, состоящий из фиксированного количества записей, причем все записи одинаковой длины, длина записи задается при создании файла. Записей не может быть больше 255 шт., каждая запись не может быть длинее 255 байт.
Средство работы — пара READ RECORD/UPDATE RECORD.
Можно использовать как абсолютную (по номеру записи) так и относительную адресацию (следующая/предыдущая, но без цикличности).
Пример использования — записная книжка на SIM-карте.

3) Cyclic - в целом — то же, что и Linear Fixed, но со следующей дополнительной функциональностью:

- при чтении: замкнутость — при использовании относительной адресации, переходя с последней на следующую запись попадаем на первую, переходя с первой на предыдущую запись, попадаем на последнюю.
- при записи: допустима только относительная адресация с обращением только к предыдущей записи. Есть даже специальная команда, только для циклических файлов, которая обновляет самую старую запись файла (после этого она становится самой новой) — INCREASE.

Первая запись всегда хранит самые новые данные, а последняя — самые старые.
Пример использования — список ранее набранных номеров. Вообще, файлы данного типа используются гораздо реже, чем файлы других типов.

В API отсутствует возможность запросить список всех файлов/папок, соответственно, для того, чтобы проверить наличие файла по заданному пути, необходимо попытаться выбрать этот файл командой SELECT и проанализировать SW (ответ карты).
В отличие от файлов в более развитых файловых системах, файлы EF в смарт-картах обладают внутренней структурой. То есть, файл EF это не поток байтов, это файл типа запись, например, так, как это определяется «файл записей» в паскале (в Delphi). Эту структуру можно индивидуально подобрать для каждого EF в соответствии с намеченным использованием. Выбранная структура файла позволяет построить элементы данных таким образом, чтобы обеспечить к ним прямой и быстрый доступ. Различают следующие основные варианты внутренней структуры файлов EF:

1. простой бинарный файл;
2. линейный файл с записями фиксированной длины;
3. линейный файл с записями переменной длины;
4. циклический файл с записями фиксированной длины.

Простые бинарные файлы являются едиными неструктурированными *блоками памяти смарт-карты*, структуру которых программа приложения может задавать удобным для нее образом. Простой бинарный файл можно рассматривать как строку байтов. Когда выполняется команда чтения или записи информации в простом бинарном файле, необходимо устанавливать смещение в байтах (от начала файла) до определенного байта (в пределах простого бинарного файла), с которого будет начинаться чтение или запись. Команда чтения или записи информации в простом бинарном файле также должна содержать счетчик или дли ну байтовой строки, читаемой из файла или записываемой в него.
Линейные файлы с записями являются файлами, содержащими более мелкие части, называемые записи (файл записей). Записи в пределах файлов нумеруются последовательно. Доступ к записям может осуществляться по их номерам или при выполнении операций последовательного чтения в прямом или обратном направлении. В файле с записями фиксированной длины все записи содержат одно и то же количество байтов. В файле с записями переменной длины каждая запись может содержать различное количество байтов. Файл с записями переменной длины требует большего времени доступа для операции чтения/записи и затрат на

администрирование. Некоторые операционные системы смарт-карт поддерживают ограниченную функциональность поиска в линейных файлах.

Циклические файлы являются линейными файлами, циклически возвращающимися к первой записи, когда читается или вносится запись, следующая за последней записью. Циклический файл удобно представить как кольцо записей. Каждая последующая операция записи в файл выполняется над следующей физической записью в кольце. Операции чтения выполняются над последней физической записью.

Операционные системы смарт-карт поддерживают обычный набор операций со всеми файлами, такие как создание, удаление, чтение, запись и обновление. Для работы с файловой системой смарт-карт определен интерфейс API в виде набора функций для выбора, чтения файлов и записи в них. С каждым файлом смарт-карты может быть связан список управления доступом. В этом списке указано, какие операции каждому из объектов, получившему доступ к карте, разрешено выполнять с файлом. Например, объект A может читать конкретный файл, но не может его обновлять, в то время как объект B может выполнять в файле операции чтения, записи и даже изменять список управления доступом.

7.4. Специальные файловые системы

Для решения задач, связанных с предоставлением доступа пользователю или программам к настройкам ядра ОС, используются так называемые специальные файловые системы. Ядро использует несколько типов специальных ФС:

tmpfs — записывает файлы в ОП. Для этого создается блочное устройство определенного объема, после чего оно подключается к папке.
procfs — хранит данные о системных процессах и ядре.
sysfs — изменяет настройки ядра ОС.

Например, ниже приведён список файловых систем, определённых в ядре linux, то есть, поддерживаемых по умолчанию. Как правило, это RAM-диски.

```
nodev    sysfs
nodev    rootfs
nodev    ramfs
nodev    bdev
nodev    proc
nodev    cpuset
nodev    cgroup
nodev    cgroup2
nodev    tmpfs
nodev    devtmpfs
nodev    debugfs
nodev    tracefs
nodev    securityfs
nodev    sockfs
nodev    bpf
nodev    pipefs
nodev    hugetlbfs
nodev    devpts

    squashfs

nodev    ecryptfs
    fuseblk
nodev    fuse
```

```
nodev    fusectl
nodev    pstore
nodev    mqueue
nodev    autofs
```

Вопросы на «засыпку»

1. Как вычислить длину дорожки на CD-диске?
2. Какие файловые системы в вашем смартфоне?
3. Строки каталога файловой системы ISO-9660 может содержать «расширения» переменной длины. Как определяется, что строка каталога закончилась и началась следующая?
4.

ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Таким образом, внутреннее содержание файловых систем имеет определённое сходство. Как было сказано в лекциях, содержанием файловых систем являются алгоритмы, из которых главную роль играют алгоритмы преобразования имён, «расшаривания», обеспечения надёжности и защиты. Эти алгоритмы обрабатывают структуры данных, определяющих устройства (носители информации). Причём, сами алгоритмы почти всегда являются частью соответствующей операционной системы (входят в состав ОС), а структуры данных конечно же пишутся на носители информации. В этих структурах данных запоминается, что же, собственно, хранится на данном носителе.

В силу многообразия операционных систем, соответственно «многообразно» и множество файловых систем.

Простые файловые системы, как правило, жёстко фиксированы, формируются однажды при создании и в дальнейшем не меняются. В них заранее (см. лекцию 7) могут создаваться каталоги и файлы определённого назначения и определённого размера и даже с фиксированными именами, которые потом в процессе эксплуатации только заполняются данными.

Более сложные файловые системы (см. лекции 4, 5, 6) уже позволяют инициативу пользователя по созданию структурных элементов — каталогов и файлов произвольного размера и именования.

Кроме того, файловые системы создаются по определённой парадигме и несмотря на создание их разными разработчиками в условиях различных концепций ОС, тем не менее содержат схожие типовые элементы в своей структуре.

Так в частности, почти всегда присутствует структурный элемент «суперблок», описывающий файловую систему «в целом». Даже в простейших файловых системах, например, на сим-картах такую роль выполняет каталог MF с именем 3F00. В файловой системе FAT, роль суперблока играет структурный элемент PBR, в ntfs — первые 16 строк MFT, в ufs/ext и других unix`овых файловых системах «суперблок» присутствует явно.

Только в простейших файловых системах нет структурного элемента «битовое поле», поскольку эти файловые системы фиксированы. В файловой системе CP/M битовое поле существует только временно, создаваясь каждый раз заново при включении ЭВМ. В файловой системе FAT таблицы FAT выполняют помимо своей основной функции адресации кластеров, также косвенно функцию битового поля. В больших файловых системах битовые поля присутствуют явно, поскольку позволяют ускорить и упростить обработку занятого/свободного пространства и строк индексных таблиц. Причём, обратите внимание, что «битовые поля» в полном объёме (покрывающих и блоковое пространство, и индексную таблицу) присутствуют только в наиболее развитых файловых системах.

Индексные таблицы, определяющие атрибуты файлов также изменяются от простого к сложному:

- в простейших файловых системах их нет вообще, в силу фиксированности файловых систем;

- в более сложной FAT индексная таблица совмещена с каталогом и при этом отделена от адресных таблиц FAT, хотя адреса кластеров с данными файла — это атрибуты файла; то есть, в файловой системе FAT нарушена концептуальная целостность файловой системы;

- в сложных файловых системах индексные таблицы выделены в отдельную сущность, содержат явно и полностью атрибуты файлов (включая адреса блоков с данными) и отделены от каталогов.

Структурный элемент «каталог» в наиболее «чистом» и явном виде - как структурный элемент файловой системы, содержащий имена объектов, хранящихся в файловой системе, - присутствует в сложных файловых системах ufs/ext/ntfs и подобных.

Кроме того, в наиболее сложных файловых системах (ufs/ext) для обеспечения повышенной надёжности и живучести существуют дополнительные структурные элементы, обеспечивающие распределённых характер хранения важных структур данных файловых систем по всему носителю (разделу).

Однако, тенденция такова, что файловые системы становятся всё более абстрактными и их API всё более скрывает от прикладного ПО сущность носителей информации. Например, если в ufs/ext/ntfs имена/адреса носителей информации присутствуют в путях к файлам, то в новейших файловых системах прикладное ПО уже не может определить, где находится файл: в разделе, на диске, томе, на каком носителе, штатном или носимом и т. д. С одной стороны, это удобно — обеспечивает переносимость программного обеспечения, но с другой стороны вызывает повышенные накладные расходы, больше ресурсов система тратит на обеспечение своего функционирования.

ИСПОЛЬЗОВАННАЯ И РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Чичев А. А., Чекал Е. Г. Операционные системы. Часть 1. Работа с операционной системой. Учебное пособие. - Ульяновск, УлГУ, 2015
2. Чичев А. А., Чекал Е. Г. Администрирование информационных систем. Часть 1. Общие вопросы. Учебное пособие. – Ульяновск: УлГУ, 2018. – 156 с.
3. Чекал Е. Г., Чичев А. А. Надёжность информационных систем. Часть 1. Учебное пособие. - Ульяновск, УлГУ, 2012
4. Чичев А.А., Чекал Е.Г. Сетевое программное обеспечение. Учебное пособие. – Ульяновск: УлГУ, 2020. – 166 с.
5. Чичев А. А., Чекал Е. Г. Операционные системы. Часть 4. Фонд оценочных средств. Учебное пособие. - Ульяновск, УлГУ, 2019.
6. Олифер В.Г., Олифер Н.А.. Сетевые операционные системы. 2-е изд. - Спб.: Питер, 2009. - 669 с.: ил.
7. В. Олифер, Н. Олифер Компьютерные сети. Принципы, технологии, протоколы. Учебник. 5-е изд. - Спб.: Питер, 2016. - 992 с: ил. - (Серия «Учебник для вузов»). ISBN: 9785496019675
8. Таненбаум Э.,, Уэзеролл Д. Компьютерные сети. 5-е изд. - Спб.: Питер, 2016
9. Таненбаум Э., Вудхалл А. Операционные системы. Разработка и реализация. 3-е изд. — Спб.: Питер, 2007. — 704 с: ил.
10. Чичев А.А., Чекал Е.Г. Стилевые конфликты и оценка стиля программирования. Сб. трудов XI международной конференции «Информационные технологии в образовании». Ч. 2. Москва, 2001. - М.: МИФИ, 2001. - С. 151-154
11. Чичев А.А., Чекал Е.Г. Подготовка ИТ-специалистов в университете. Ученые записки УлГУ, серия "Математика и информационные технологии" – Вып. 1(4). - Ульяновск: УлГУ, 2012. – 226с. - С. 278-286.
12 Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 405 с.
13. Ногл М. TCP/IP. Иллюстрированный учебник - М.: ДМК Пресс, 2001. - 480 с.: ил. - ISBN 5-94074-044-8
14. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.:Питер, 2015. — 1120 с.: ил. — (Серия «Классика computer science»). ISBN 978-5-496-01395-6
15. Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. - Спб.: Питер, 2018. — 816 с: ил. - ISBN 978-5-496-00337-7
16. URL: //top500.org — дата доступа 07.12.19